iDSL: Automated Performance Evaluation of Service-Oriented Systems

Freek van den Berg[1](✉), Boudewijn R. Haverkort[1], and Jozef Hooman[2,3]

[1] Design and Analysis of Communication Systems,
University of Twente, Enschede, The Netherlands
{f.g.b.vandenberg,b.r.h.m.haverkort}@utwente.nl

[2] TNO-ESI, Eindhoven, The Netherlands

[3] Radboud University, Nijmegen, The Netherlands
j.hooman@cs.ru.nl

Abstract. Service-oriented systems interconnect with other systems in a time critical manner, making their performance vital. For this purpose, we propose an automated performance evaluation approach for service-oriented systems which includes both performance measurement and prediction. The approach makes use of the iDSL language, a domain specific language tailored to modeling service-oriented systems, and the iDSL toolchain to evaluate iDSL models, as follows. First, discrete-event simulation yields many performance artifacts, e.g., latency breakdown charts, cumulative distribution graphs, and latency bar charts. Second, timed automata-based model checking yields absolute latency bounds. Third, probabilistic timed automata-based model checking leads to exact latency distributions for each service. We successfully validated our approach; several case studies on interventional X-ray systems displayed similar measured and predicted outcomes.

1 Introduction

An embedded system is a computer system that has a dedicated function within a larger system, often with real-time computing constraints [16,27]. Today, the majority of the commonly used devices are embedded systems, ranging from simple digital watches, to complex medical machines [19]. An embedded system is frequently used to perform safety critical tasks, which makes their malfunctioning prone to serious injury and fatalities, such as with medical systems. An embedded system interacts with its environments in a time critical way. Its safety is therefore predominantly determined by its performance, which is expressed in terms of response times, resource utilizations and queue sizes.

Many current practices only address the performance at the end of the development trajectory and only resort to tuning of the performance until the system is "good enough" [21]. Contrarily, we advocate that each design decision during system development should be evaluated for performance immediately, as it can have an increasing impact on performance later on. This prevents unexpected performance issues that are hard and costly to fix, especially the ones

DOI: 10.1007/978-3-319-68270-9_11

that are detected late. On top of that, it is recommended to make use of performance predictions, which can provide early insight in the performance of different design alternatives, without having to realize an actual system yet. We claim that prediction-based performance evaluation should be an integral part of the design of complex embedded systems [18].

Good performance is hard to achieve, because embedded systems come with increasingly heterogeneous, parallel and distributed architectures [18]. At the same time, they are designed for many product lines and different configurations, which gives rise to many potential designs. Moreover, accurately predicting performance characteristics of embedded systems is hard, since the real system does not exist yet. Once the system has been built, measurements to gain insight in the performance tend to be expensive, because they require the system to run for quite some time, e.g., to detect rare performance outliers.

To narrow our scope, we constrain ourselves to so-called *service-oriented systems* [15,23–26], a special class of embedded systems that have the following four properties. First, service-oriented systems provide services to their environment, accessible via so-called service requests. Second, a service request leads to exactly one service response. Third, individual service requests are isolated from each other in a service-oriented system and do, therefore, not affect each other's functionality. Fourth, service requests may negatively affect each other's performance by competing for the same resource in the service-oriented system.

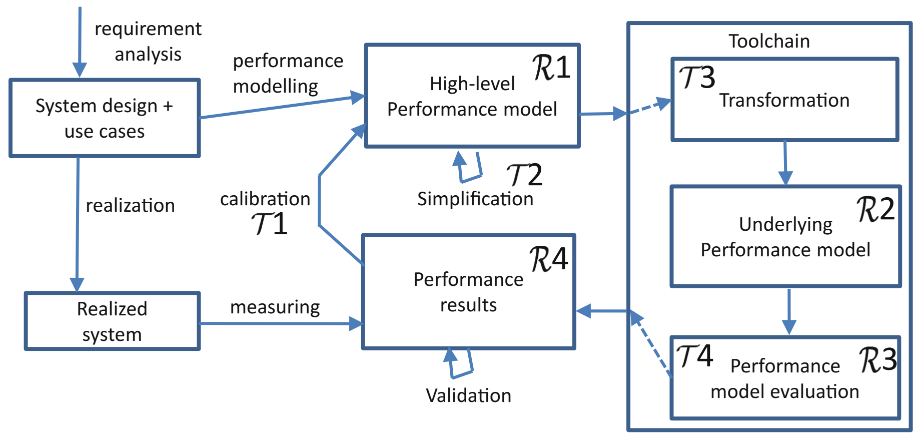


Fig. 1. System development including performance measurement and prediction.

We propose a framework for the performance evaluation of service-oriented systems (see Fig. 1), comprising performance measurement *and* prediction.

Performance measurement starts with the realization of a "system design + use cases", a blueprint of the system to be realized, resulting into a "realized system" via a realization. Performing measurements on this system then yield "performance results". Performance measurement comes with two downsides: (i) realizing a system is (often) very costly, let alone realizing many different systems only for testing purposes; and, (ii) since exhaustive performance measurement is impossible, assessing the performance of rare but important events is difficult.

Performance prediction starts with modeling the performance of a "system design + use cases", which may include the use of existing "performance results", i.e., measurements, for model calibration. The resulting "high-level performance model" enters the "toolchain" in which it is transformed into a "underlying performance model", and subsequently evaluated for performance, yielding "performance results". In this paper, performance prediction relies on measurements and can therefore only be used in addition to performance measurement. Also, it is model-based and thus inherently inaccurate. On the upside, however, a "high-level performance model" may represent many system designs and performing measurements on them can often be done at high speed. Hence, performance prediction is suitable for quickly evaluating the performance of many designs and thus enables design space exploration [5,11].

In this paper, we focus on performance prediction only. We do thereby assume that measurements for calibration purposes are readily available.

In the following, we address four requirements that a performance prediction approach should meet. For this purpose, Fig. 1 has been augmented with labels $\mathcal{R}1$, $\mathcal{R}2$, $\mathcal{R}3$ and $\mathcal{R}4$, respectively. We elaborate on them, as follows:

- $\mathcal{R}1$ A high-level performance model should be expressive, yet concise. The model should allow for the use of different evaluation techniques. An integrated Development Environment (IDE) and/or graphical user interface (GUI) should be provided to ease modeling. The model should support calibration based on measurements and the Y-chart philosophy by supporting, and separating, the applications, platforms, and mappings. Finally, mechanisms such as compositionality, layers and hierarchies, and/or classes, are called for.
- $\mathcal{R}2$ The underlying performance model should make it easy to analyze complex high-level performance models and also support the enabling and disabling of different properties, such as nondeterministic choices.
- $\mathcal{R}3$ The performance evaluation process should be **fully** automated; a system designer creates a model, which is evaluated without any user interaction, and results are automatically returned. This includes Design Space Exploration [5,11] and post-processsing steps, e.g., visualizations. Also, multiple modes of analysis should be supported for models of reasonable complexity. Besides discrete-event simulations for quick results, model checking should be supported for more accurate results. Finally, evaluating the model should take a limited amount of time and scale well for models of complex systems.
- $\mathcal{R}4$ Performance evaluations should lead to various results types, e.g., utilizations, latency breakdown charts, and latency distributions. Whenever possible, the results should be presented visually for easy human interpretation.

In previous work [22], many toolsets that support the performance evaluation process have been compared regarding these requirements; often their level of automation is limited, their models tend to be at a lower level of abstraction as called for, only one way of analysis is supported, and results are not visualized.

This paper's remainder is organized, as follows. Section 2 reveals so-called interventional X-ray systems, which are medical systems. Section 3 introduces the high-level iDSL language, followed by the iDSL tool chain in Sect. 4. Section 5 provides an extensive case study. Finally, Sect. 6 concludes the paper.

2 Interventional X-ray Systems

For a running example, we introduce interventional X-ray (iXR) systems, which are medical imaging systems that enable minimally-invasive surgeries.

An iXR system consist of a number of parts (as depicted in Fig. 2), as follows. It is used to assist a surgeon while performing surgery, during which a patient lies on a **table**. The iXR system displays a continuous stream of images of (the inside of) a patient on a display. These images are based on X-ray beams, generated in the **arc** and caught by the detector, whose task it is to extract raw images from X-ray beams. Via the control panel, the surgeon can move the arc and table in various ways and thereby change the angle of the recorded images of the patient, which are shown continuously on the display.

An iXR system needs to support different settings so that it can be customized for a specific patient, surgeon and procedure, for instance: (i) mono- or biplane, i.e., using either one or two X-Ray beams to generate and detect images, yielding 2D or 3D images; (ii) image resolution, e.g., images of 512^2, 1024^2 or 2048^2 pixels; and, (iii) image frame-rate, e.g., 5, 10 or 25 images per second.

Image Processing (IP) is an important subsystem of an iXR system. It turns raw X-ray images into high quality ones, in real-time; IP retrieves unprocessed images from the X-ray **detector**, processes them to enhance their quality, and delivers them to the **display** to be seen by the surgeon (see Fig. 2). IP comprises different kinds of operations, e.g., detecting so-called dead pixels, reducing spatial and temporal noise, and preparing an image for a particular display.

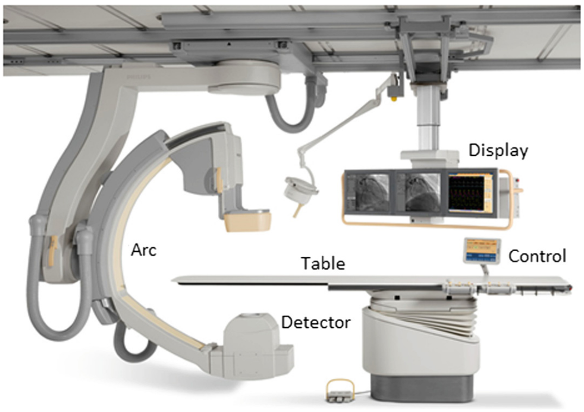


Fig. 2. The main parts that constitute an interventional X-ray system

IP is a trade-off between (i) constant quality and frame-rate of the images, (ii) average throughput, latency and jitter of individual images, (iii) amount of X-Ray a patient and surgeon get exposed to during a treatment, and (iv) required computational resources to process images.

The safety of IP is mainly determined by performance, viz., a surgeon needs to continuously receive high quality images to perform surgery on a patient. Hence, the image latency, the time between an image arriving on the detector and appearing on the display, needs to meet a strict requirement. Literature suggests an average latency below 165 ms for proper hand-eye coordination [12].

3 The iDSL Language

In this section, the iDSL language, which forms a conceptual model of a service-oriented systems, is defined [22,25]. It comprises six sections (see Fig. 3).

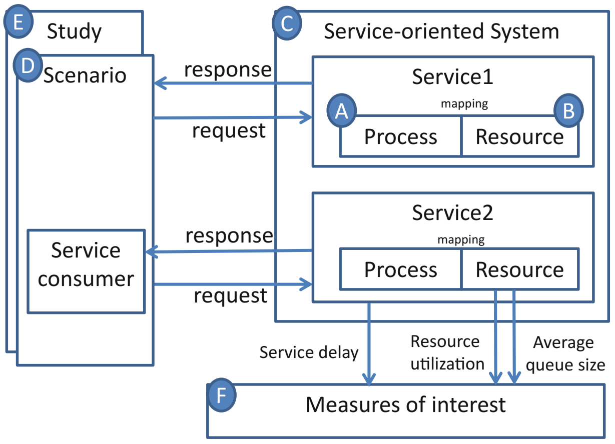


Fig. 3. A conceptual model of a service system.

A *service-oriented system* (see Fig. 3-C) provides one or more services to one or more *service users* (exterior to the service-oriented system), viz., a *service user* sends a request for a specific service at a given time, after which the system responds with some delay. A *service* is an entity that performs functions ranging from simple requests to computationally expensive processes.

Service-oriented systems do not only need to return the right answers to requests, but also face stringent performance constraints, e.g., the system has to reply to a request within a certain time, often referred to as *latency*. Service-oriented systems can particularly be hard to analyze when they handle many service requests in parallel, for multiple kinds of services, in a real-time manner.

A *service* is decomposed into one or more processes, resources and a mapping, in line with the Y-chart philosophy [14]. A *process* (see Fig. 3-A) decomposes high-level service requests into atomic tasks, each assigned to a resource via the *mapping* (not shown in the figure). Hence, the mapping connects the process to resource it relies on. A *resource* (see Fig. 3-B) is capable of performing one atomic task at a time. When multiple services are invoked of which the resources they rely on overlap, contention may occur, making performance analysis hard.

A *scenario* (see Fig. 3-D) comprises a number of invoked service requests over time to observe the performance behavior of the service system in specific circumstances over time. Service requests are functionally independent of each other, i.e., service requests do not affect each other's functional outcomes, but may affect each other's performance negatively due to contention.

A *study* (see Fig. 3-E) is a set of scenarios to be evaluated, so as to derive the system's underlying characteristics. Within a study, a *design space* is an efficient way to describe a large number of similar scenarios.

Finally, a *measure of interest* (see Fig. 3-F) defines an interesting performance metric, given a system and scenario, e.g., latencies and queue sizes.

In this section's remainder, we illustrate the meaning of the iDSL language via a running example of a so-called biplane iXR system (see Sect. 2).

The high-level Process decomposes a service into several atomic tasks, represented by a recursive data structure with layers of sub-processes. At the lowest abstraction level, an atomic task specifies a workload, e.g., some CPU cycles.

The example process of Table 1 combines hierarchies (curly brackets), sequential compositions (*seq*) and atomic tasks (*atom*). At its highest level, it consists of a sequential task "image_processing_seq" that decomposes into an atomic task "pre-processing" with (fixed) load 50, a sequential task "image_processing" and an atomic task "post-processing" with load 25. In turn, the sequential task "image_processing" decomposes into three atomic tasks named "motion compensation" with load 44, "noise reduction", and "contrast" with load 134. The load of "noise reduction" is drawn from a uniform distribution on [80, 140].

As in [3], iDSL also supports process algebra constructs for parallelism (*par*), nondeterministic (*alt*) and probabilistic choice (*palt*), mutual exclusion (*mutex*) to permit only one process instance to enter a certain subprocess at a time, and design alternative (*desalt*) to implement a subprocess that varies across designs.

Table 1. The code of an iDSL process

```
Section Process
  ProcessModel image_processing_application
    seq image_processing_seq {
      atom image_pre_processing load 50
      seq image_processing {
        atom motion_compensation load 44
        atom noise_reduction load uniform(80:140)
        atom contrast load 134 }
      atom image_post_processing load 25 }
```

The high-level Resource is decomposed into a number of atomic resources (*atom*) via different layers of decomposable resources (*decomp*). Each atomic resource has a constant rate that specifies how much load it can process per time unit, e.g., the number of CPU cycles per second. The example resource "image_processing_-decomp" of Table 2 is a composite resource which consists of two atomic resources, i.e., a "CPU" with rate 2 and a "GPU" with rate 5.

Table 2. The code of an iDSL resource

```
Section Resource
  ResourceModel image_processing_PC decomp
    image_processing_decomp { atom CPU rate 2, atom GPU rate 5 }
```

Table 3. The code of an iDSL system

```
Section System
  Service image_processing_service
    Process image_processing_application
    Resource image_processing_PC
     Mapping assign ( image_pre_processing, CPU )
       ( motion_compensation, CPU )( noise_reduction, CPU )
       ( contrast, CPU )( image_post_processing, GPU)
```

Table 4. The code of an iDSL scenario

```
Section Scenario
  Scenario image_processing_run
    ServiceRequest image_processing_service at time 0, 400, ...
    ServiceRequest image_processing_service
      at time dspace("offset"), (dspace("offset")+400), ...
```

The high-level System provides one or more services to its environment. The example system of Table 3 comprises one service which decomposes into a process (see Table 1), resource (see Table 2) and a mapping. This decomposition makes it easy to change a process and/or resource part of a service, e.g., to apply Design Space Exploration (DSE). Finally, the mapping assigns atomic task "image_post_processing" to "GPU", and the other atomic tasks to "CPU".

The high-level Scenario comprises several service requests to a system, each for a given service and at a certain time. The example *biplane* iXR system of Table 4 has two services types, viz., for frontal and lateral IP. They are modeled similarly, i.e., using the example service of Table 3. Frontal IP requests occur with fixed inter-arrival times of 400, without an initial delay. Lateral IP requests also have inter-arrival times of 400, but the initial delay is design dependent,

Table 5. The code of an iDSL study

```
Section Study
  Scenario image_processing_run
    DesignSpace ("offset" "0" "20" "40" "80" "120" "160" "260")
```

Table 6. The code of an iDSL measure

```
Section Measure
  Measure ServiceResponseTimes using 1 run of 250 requests
  Measure ServiceResponseTimes absolute
```

viz., the *dspace* operator with parameter "offset" refers to dimension "offset" in the design space (see Table 5), which varies from 0 till 260.

The high-level Study characterizes a set of designs to compare. A design space is a shorthand way to specify many designs; it consist one or more dimensions that each have several possible values. A design provides a unique valuation for each dimension. The example study of Table 5 encompasses a design space with one dimension "offset" that comprises seven values. The dimension is used to vary the degree of concurrency between both services in Table 4.

The high-level Measure defines, given a system and scenario, the metrics the system designer is interested in and how they are obtained.

The example measure of Table 6 consists of two measures that both return service response times. To this end, the first measure uses 1 discrete-event simulation run of 250 requests (see Sect. 4.4, discrete-event simulation) which provides insight in resource utilizations and latency breakdowns in one go. The second measure employs an iterative model checking approach (see Sect. 4.4).

4 The iDSL Toolchain

The iDSL toolchain takes an iDSL model (of Sect. 3) as input and automatically generates a wide array of performance artifacts. For this purpose, the iDSL toolchain subsequently executes the following four steps: (i) calibrate the model on the basis of measurements (see Sect. 4.1); (ii) simplify the model (see Sect. 4.2); (iii) transform the model into a low-level Modest model (see Sect. 4.3, [8–10]); and, (iv) evaluate the performance of the model (see Sect. 4.4). In Fig. 1, these steps are labeled $\mathcal{T}1$, $\mathcal{T}2$, $\mathcal{T}3$ and $\mathcal{T}4$, respectively.

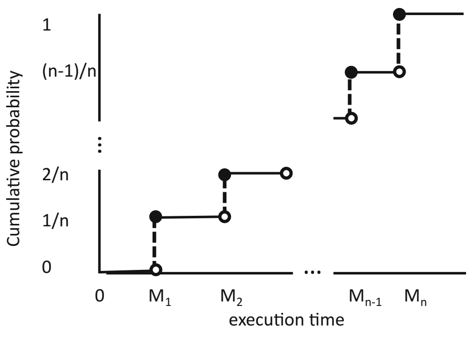


Fig. 4. The Empirical Distribution Function for n numerically-sorted measurements $m_1, m_2, \ldots, m_{n-1}, m_n$

4.1 Calibrating the Model on the Basis of Measurements

In Sect. 3, an atomic task load in a iDSL process (see Table 1) is either fixed or uniformly drawn from a certain interval. Next, we propose an Empirical Distribution Function (EDF) as a third possibility to enable model calibration on the basis of measurements. In Table 7, we observe that an EDF load has one parameter and comes in two flavors. Atomic task "edf_values" is initiated with a sequence of measurements values, whereas "edf_files" refers to an external file containing measurements.

The EDF for n numerically-sorted measurements $m_1, m_2, \ldots, m_{n-1}, m_n$ (see Fig. 4) is a step function that jumps up by $\frac{1}{n}$ at each of the n data points. Optionally, iDSL provides EDF prediction during which EDFs are predicted for designs for which no measurements have been performed, on the basis of existing EDFs. This is carefully explained in [22,26], but beyond the scope of this paper.

Next, we show how model simplification is applied to an atomic task with an EDF load (in Sect. 4.2), after which it is transformed into a palt-construct as part of the transformation to Modest (in Sect. 4.3).

Table 7. The code of a small iDSL process with an EDF

```
Section Process
  ProcessModel image_processing_application seq {
     atom edf_values load EDF with values 6 8 10
     atom edf_files load EDF from file "measurements.dat" }
```

4.2 Simplifying the Model

Generally, an iDSL model is often too hard to analyze, especially when the iDSL processes have variable loads. To this end, iDSL comes with two model simplification techniques for EDFs in the iDSL process, viz., the clustering of loads and changing the model time unit.

The clustering of loads is applied to atomic processes that are defined as an EDF. Measurements are clustered into a number of clusters using K-means clustering [13]. Each cluster is summarized by the smallest interval of nondeterministic time containing all its measurements. These intervals are combined via a probabilistic choice, thereby reducing the number of probabilistic alternatives.

Figure 5 presents a small example based on three measurement values 6, 7 and 18. Figure 5(a) shows the original EDF, which assigns an equal weight of $\frac{1}{3}$ to each of the 3 measurements. When the number of given clusters is at least the number of measurements, this original EDF is returned since each measurement is assigned to its individual cluster. Figure 5(b) displays the result of K-means clustering with 2 clusters, viz., measurements 6 and 7 are grouped in one cluster due to their proximity, and 18 in the other. Hence, 6 and 7 are represented by a nondeterministic time interval, graphically depicted as a grey area covering

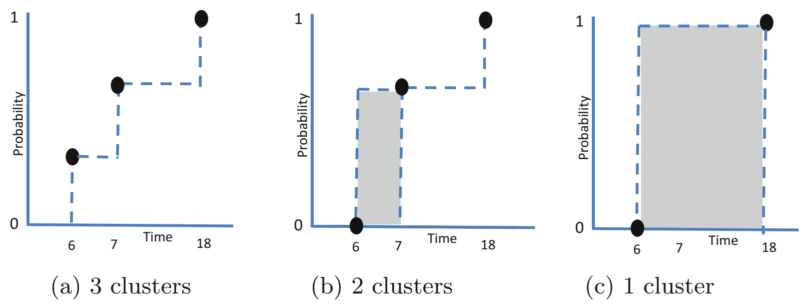


Fig. 5. EDFs based on measurements 6 µs, 7 µs and 18 µs that are simplified using K-means clustering.

time range [6 : 7], and probability range $[0 : \frac{2}{3}]$. This grey area represents an ambiguity, viz., all distributions that go through this area are possible. The result is accurate, which means that the real distribution goes through this area. [23] quantifies this ambiguity. Finally, Fig. 5(c) shows the one cluster case. All measurements end up in one cluster, yielding a nondeterministic time range [6 : 18] and probability range [0 : 1].

Changing the model time unit increases the global time unit of the iDSL model. It is, among others, applied to all EDF functions in the model: (i) the measurements are divided by the chosen time unit and rounded to the nearest integer value; (ii) performance evaluation is applied; and, (iii) the results are multiplied by the time unit again. Bigger time steps reduce the model complexity, whereas rounding reduces precision. In [23], this loss of precision is quantified. Note that rounding can both lead to conservative or overestimated results.

Figure 6 shows an example, again for measurements 6, 7 and 18. Figure 6(a) shows the case of time unit = 1, i.e., dividing measurements by 1 does not introduce rounding errors. Figure 6(b) highlights the case for time unit = 6. Measurements 6 and 18 are not rounded by being multiples of 6, but measurement 7 induces a rounding error, viz., an integer division of 7 by 6 followed by a multiplication by 6 yields 6 instead of 7. Effectively, measurement 7 is replaced by a 6,

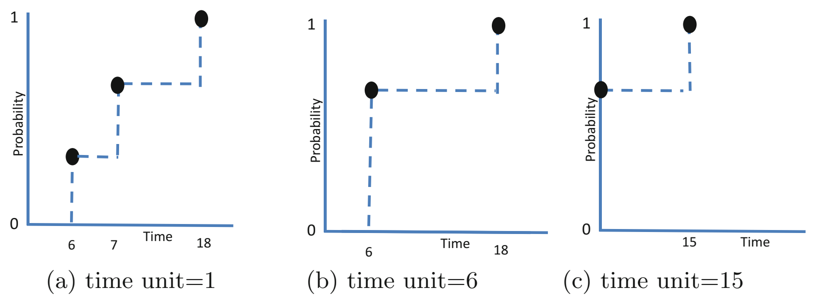


Fig. 6. EDFs based on measurements 6 µs, 7 µs and 18 µs that are simplified by increasing the time unit.

yielding two 6 and one 18 values. Figure 6(c) the shows case for time unit = 15. Measurements 6 and 7 are both rounded to 0, and measurement 18 becomes 15.

Finding the right abstraction level of the model is achieved by systematically benchmarking iterations of MCSTA for models with different combinations of clusters and time units. It is the objective to find a model that computes fast enough and at the same contains enough level of detail. [22,23] describes the algorithm in detail.

4.3 Transforming the iDSL Model into Equivalent Modest Models

We explain how an iDSL model transforms into a set of Modest models [8–10] (as graphically depicted in Fig. 7). [25] provides a concrete example. On top, a Modest model comprises a parallel execution of interacting processes, i.e., services, resources and generators, implemented using a par-construct. This similar to a system in LOTOS [3].

For each ProcessModel in the iDSL process, a similar Modest process is generated. To this end, there are two types of processes. First, a compound process contains one operator, e.g., par, seq, alt and palt, and recursively refers to subprocesses. Furthermore, an atomic process with an EDF load is transformed into a palt-construct: an alternative is created for each jump in the cumulative distribution function with a weight corresponding to the jump size. For instance, Fig. 4 conveys a jump from $\frac{1}{n}$ to $\frac{2}{n}$ at time M_2, which translates to a palt-alternative with weight $\frac{2}{n} - \frac{1}{n} = \frac{1}{n}$ and time M_2. Note that each alternative is an atomic process. Second, atomic processes signal their ID and a load to a fixed resource queue, viz., the one defined in the mapping, and wait for a result.

For each ResourceModel in the iDSL resource, a Modest resource_queue and resource process are generated, which both repeat forever (as indicated by the repeat symbol) for FIFO scheduling. A resource_queue either receives an ID and load from a atomic process (a buffer addition), or forwards an ID and load to its resource (a buffer removal), each iteration. In turn, a Modest resource waits for a resource_queue to provide an ID and load, processes it using a delay (as indicated by the stopwatch), and returns the result to the atomic process with the given ID. A resource_queue is not generated in case of nondeterministic scheduling, since the order at which requests arrive is not relevant. In this case, the ID and a load an atomic process signals is directly connected to the resource process. We stress that other scheduling methods are not yet supported.

For each Service in the iDSL system, one or more Modest services are generated. Each service alternately waits for an incoming request trigger and activates the process that corresponds to the service. The number of created Modest services decide how many service instances the system can handle simultaneously, which is one by default to have a simple model. However, it can be overridden by the "numinstances" keyword. Moreover, the service mapping is used to connect the atomic processes and resource_queues (see the green circle in Fig. 7).

For each ServiceRequest in the iDSL Scenario, a Modest init_generator is generated that generates an initial delay, followed by a call to a Modest generator process that forever triggers a service at fixed inter-arrival times.

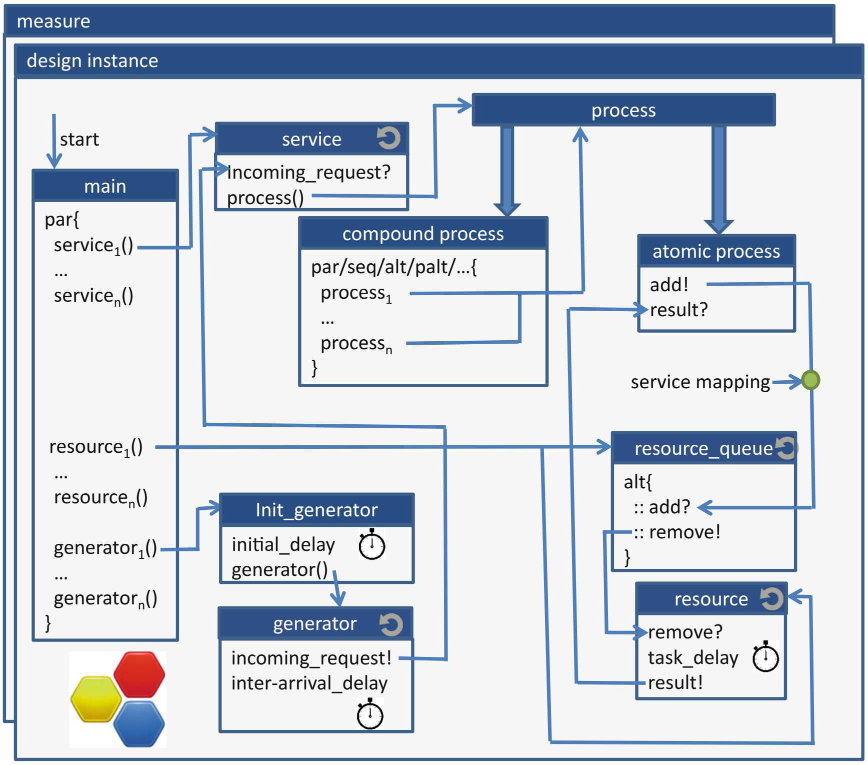


Fig. 7. The Modest processes that are generated given an iDSL model. (Color figure online)

Finally, similar Modest models are created for each design in the iDSL study and for each measure in the iDSL measure, as follows. First, a design instance provides a valuation for each dimension. It is used, among others, to replace the *dspace* operator by the actual value of the respective dimension in arithmetic expressions, e.g., the service request times in Table 4. Second, a measure specifies how a Modest model is analyzed. This requires measure specific adjustments to be made to the model (see Sect. 4.4), e.g., turning real values into integers.

4.4 Evaluating a Modest Model to Yield Performance Artifacts

In this section, Modest models of the previous section are evaluated for each design, using different evaluation techniques. Besides discrete-event simulation, there are four ways of model checking, viz., TA-based, PTA-based, and efficient PTA-based, and efficient & scalable PTA-based. Each technique comprises three execution steps: (i) the Modest model is modified to be compatible with the given technique; (ii) a Modest tool is applied to the Modest model at least once yielding performance numbers; and, (iii) post-processing turns the performance numbers into artifacts. Next, we explain these steps for each technique.

Discrete-event simulation yields latencies of services, subprocesses and resource utilizations. Latencies are obtained by enclosing each service and subprocess with stopwatches. An additional service counter and properties for each subsequent request of a given service then make it possible to retrieve individual latencies for a service. A resource utilization is obtained by adding a counter to a resource that keeps track of the total time it is processing. This counter value is divided by the total running time of the system, which is implemented as a global counter. For each run, MODES of the Modest toolset [10] is used once to perform a discrete-event simulation on the Modest model. MODES is instructed to use an as soon as possible (ASAP) scheduler for time, and uniform resolution for nondeterminism choice. This is a pragmatic and commonly used choice that does not need to reflect the real underlying structure [4].

Post-processing yields three performance artifacts. First, a latency bar chart (see Fig. 11(c)) is generated using GNUplot [20], which visually displays succeeding latency times.

Second, a latency breakdown chart (see Fig. 11(a)) conveys the static process structure of a service extended with its dynamics, i.e., latencies and utilizations. The graph structure is derived by recursively traversing the iDSL process and resource. It is augmented with placeholders in one go wherever performance numbers are needed. Next, these placeholders are replaced by the relevance Modest properties, after which GraphViz [6] renders the visualization.

Third, a cumulative distribution graph (see Fig. 11(b)) displays latency times for different designs. Hence, they are convenient to get insight in the consequences of certain design decisions. To this end, the latency values of the different designs are gathered, combined, and turned into a plot using GNUplot [20].

TA-based model checking yields absolute service latencies. To make the model finite, real values that concern time (including loads and rates) are rounded to their nearest integer values. Additionally, probabilistic choices and infinite distributions are replaced by nondeterministic choices. Latencies of services are obtained by enclosing each service with stopwatches that reset after registering one latency. To reduce the state space size, no service counter is added; we do not retrieve which latency is the maximum one. Combined, this leads to a finite, decidable model. TA-based model checking is performed using MCTAU [1]. Via a binary search algorithm of [25], i.e., recursive function lb (in Table 8), an initial range of possible values is halved iteratively until one value

Table 8. Function lb: compute lower bounds, pseudo code

```
lb ([lbound:ubound]){
    if (abs(ubound-lbound)<=1) return lbound          // case a
    check_value=(lbound+ubound)/2
    UPPAAL (p = probability(latency<check_value))
    if ( p=0 ) lb (check_value,ubound)                // case b
    else lb (lbound,check_value) }                    // case c
```

remains. This initial range is $[0 : n]$, where n is a deliberate overestimation of the latency. Each iteration, one of the following three cases occur: (a) there is only one possible value left, which is returned; (b) model checking conveys that the probability that the value is in the lower half of the values is 0 in which case the upper half of the values is returned; or, (c) the lower half of the values is returned. The complexity of the algorithm is $\mathcal{O}(log(n))$, where n is the chosen overestimation.

PTA-based model checking yields exact service latency distributions for each service; a Modest model is created for each service and both the minimum and maximum probability to compute the latency distribution of that particular service. In each Modest model, only the process of the given service is enclosed by stopwatches that record latencies of the service requests. The Modest models have one parameter time $t \in \mathbb{R}_{\geq 0}$, and return a probability p: the probability that the service completes within time t.

In iDSL, however, a service leads to an infinite stream of service requests, each with its individual latency. Ideally, the average of this infinite stream of latencies is a measure for the performance of the whole service. Put formally:

$$P_{\Omega}(t) = \lim_{k \to \infty} \frac{1}{k} \sum_{n=1}^{k} P_n(t), \tag{1}$$

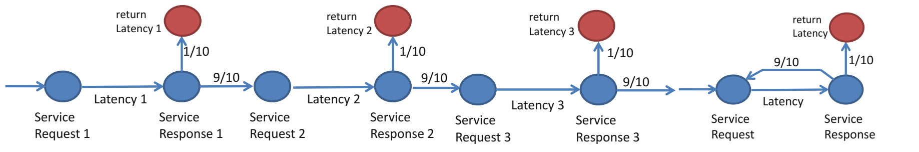
where $P_{\Omega}(t)$ is the combined probability, n the service request number, t the latency time, $P_n(t)$ the probability that service request n finishes within time t.

This infinite sum cannot be directly computed. Instead, the computable *geometric distribution* [17] is proposed that is capable of detecting an absolute maximum latency, weighing service requests in an exponentially decreasing way:

$$P_{\Omega}(t) = \sum_{n=1}^{\infty} (1-\rho)^{n-1} \rho \, P_n(t), \tag{2}$$

where $\rho \in (0 : 1)$ is the geometric distribution parameter. The distribution is similar to (1), for $\rho \approx 0$ and capable of finding the absolute latencies.

In Modest, the geometric distribution is implemented as a binary probabilistic choice every time a service request completes (as depicted in Fig. 8(a)): either the currently measured latency is returned, with probability ρ ($\frac{1}{10}$ in the figure), or the next service request is evaluated, with probability $1-\rho$ ($\frac{9}{10}$ in the figure).



(a) with a service request counter (b) without a counter

Fig. 8. Binary probabilistic choices induce the geometric distribution

The geometric distribution is memoryless, i.e., the binary choice does not rely on the service request number. It can thus be represented as a single reoccurring service request (as depicted in Fig. 8(b)). The figure conveys that a lower ρ-value yields a more complex model and more precise results, viz., for small values of ρ, the probability that state "Service Response" is followed by many occurrences of "Service Request" increases. In this paper, we empirically choose $\rho = \frac{1}{10}$; it leads to a state space that is large enough to deliver a reasonable amount of accuracy and which is moreover practically handleable.

The algorithm to compute a latency distribution of a service [22,24] comprises three steps in which MCSTA, the explicit-state model checker for STA of the Modest toolset, is iteratively applied, as follows. First, the *initial scan* is used to obtain an upper bound on the latency. Second, the binary *lower & upper bound search* are binary searches, similar to TA-based model checking, to obtain a exact lower an upper bound. Third, the whole distribution is obtained by computing all values between the bounds in a *brute force* way.

Efficient PTA-based model checking provides the same functionality as its inefficient counterpart, but in a more efficient manner [22,23]. The efficiency gain is the result of executing three lightweight techniques initially as shown in Fig. 9. Besides a so-called basic estimate function, we reuse the already introduced discrete-event simulation and four TA-based model checking techniques.

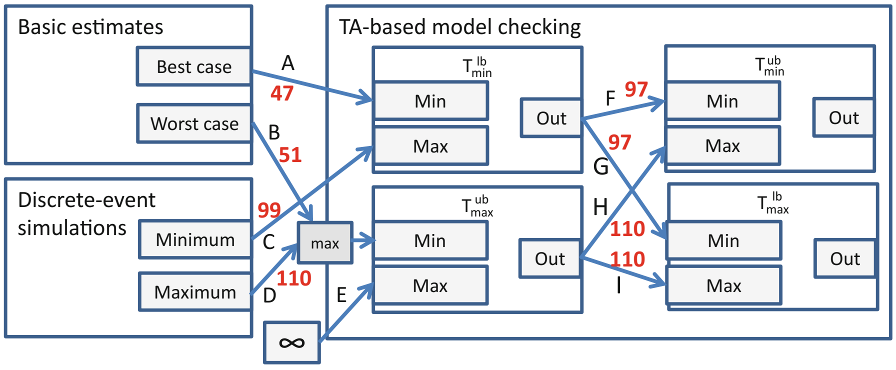


Fig. 9. The dataflow of three lightweight techniques. On the edges, execution times for service Frontal IP and offset = 0 are shown.

A basic estimate function returns, given a iDSL process, an optimistic but possibly inaccurate bound of either the minimum (maximum) latency. It is easy to compute. The result is optimistic because the concurrency between services and processing steps for resources are not taken into account. Hence, the best case can be used as a minimum for T^{lb}_{min} (**A**), a lower bound for T_{min}, and the worst case for T^{ub}_{max} (**B**), an upper bound for T_{max}. Table 9 conveys the recursive definition of the best case, as follows: (i) for an atomic process, the taskload is returned; (ii) for a (probabilistic) alternative process, the minimum of all recursively evaluated children processes is returned; (iii) for a parallel process, the maximum of the evaluated children processes is returned; and, (iv) for a sequential process, the sum of the evaluated children is returned. The worst case is defined analogously, but returns the maximum in case of a (probabilistic) alternative process.

Table 9. The recursive definition of the basic estimate (best case) function

BE: Basic estimate (best case) function

```
BE atom{p_1}          = p_1.taskload
BE alt{p_1,...,p_n}   = MIN { BE x | x in {p_1,...,p_n} }
BE palt{p_1,...,p_n}  = MIN { BE x | x in {p_1,...,p_n} }
BE par{p_1,...,p_n}   = MAX { BE x | x in {p_1,...,p_n} }
BE seq{p_1,...,p_n}   = BE p_1 + BE p_2 + ... + BE p_n
```

Discrete-event simulations display average behavior, which means that the minimum outcome of all runs can be used as a maximum for T^{lb}_{min} (**C**), and the maximum outcome as a minimum for T^{ub}_{max} (**D**). Using the maximum of **B** and **D** for the minimum of T^{ub}_{max}, makes the range of T^{ub}_{max} as small as possible.

TA-based model checking T^{lb}_{min} and T^{ub}_{max} provide an absolute minimum and maximum, i.e., regardless of how nondeterminism is resolved, respectively. They are used as a minimum (**F+G**) and maximum (**H+I**) for t^{ub}_{min} and t^{lb}_{max}.

Efficient PTA-based model checking comprises five steps: (i) compute the basic estimates; (ii) perform multiple discrete-event simulation runs; (iii) perform TA-based model checking t^{lb}_{min} and t^{ub}_{max}; (iv) perform TA-based model checking t^{ub}_{min} and t^{lb}_{max}; and, (v) execute brute force PTA-based model checking.

Efficient and scalable PTA-based model checking is similar to the previous technique, but is applied to a model that is simplified using the algorithm of the Sect. 4.2. Overall, it aims to deliver a practical compromise between the amount of needed memory, amount of wall clock time, and quality of the results. In Sect. 5.2 (results), we illustrate the concrete efficiency gain.

5 Case Study on Interventional X-ray Systems

This section conveys two experiments in which various performance artifacts are returned (in Sect. 5.1) and exact results are computed (in Sect. 5.2).

5.1 Experiment I: Retrieving a Wide Array of Performance Artifacts

Experiment I focuses on generating many performance artifacts. This takes its toll on the high-level model quality which is simple, and moreover limits performance evaluation, viz., primarily discrete-event simulations are performed. For the sake of efficiency, the running example as introduced in Sect. 3 is reused.

Transformation. The resulting Modest code comprises a parallel process at its highest level, which contains service "image_processing_service", resources "CPU" and "GPU" that run forever, and two generators that call "image_-processing_service". The service waits for incoming requests of either of the generators and triggers the process, similar to the iDSL process, in return. Atomic processes in the process call the respective resources, which perform a delay. Since nondeterministic scheduling is employed, the resources have no queues.

Evaluation. The iDSL measure (see Table 6) contains two measures, as follows. First, discrete-event simulation yields a single MODES execution that leads to latencies and utilizations in one go.

Second, TA-based model checking includes rounding the real values to integers. Since all but one values of the loads and rates are integers already, the model is not affected by this step. The uniform choice of atomic task "noise_reduction", however, is turned into a nondeterministic equivalent. Stopwatches are added to measure latencies. Given this model, a lower and upper bound latency are obtained via the binary search algorithm of Sect. 4.4 (TA-based model checking).

Results come in various kinds, as follows. First, the latency bar chart for offset = 0 of Fig. 11(c) conveys that the latencies vary much, i.e., between 200 and 380, as a result of extreme concurrency. This variation is less for other offsets.

Second, the latency breakdown chart for offset = 0 of Fig. 11(a) illustrates how the overall latency is dispersed over its subprocesses. Tasks "Noise_reduction" and "Contrast" account for 71% of the total latency. The utilization of "CPU" of 0.83 is high, but not alarming. The utilization of "GPU" is low, viz., 0.025.

Third, the cumulative distribution graph of Fig. 11(b) displays the cumulative latency functions for seven designs with offsets varying from 0 to 200. As anticipated, the offsets and latency times are negatively correlated, i.e., a smaller offset induces that the execution of services overlap more (see Table 13) and thus display more concurrency. In turn, this leads to a higher latency.

Fourth, Fig. 10 conveys, for a system with one service and obtained via TA-based model checking, the minimum and maximum absolute latency, viz., 159 and 189, respectively. It also shows a CDF of the same system based on discrete-event simulation. We observe that the bounds are valid, i.e., $s(159) = 0$ and $s(189) = 1$, and strict, i.e., $s(159+\epsilon) > 0$ and $s(189-\epsilon) < 1$, where $\epsilon > 0$ and $s(n)$ is the probability that a latency equal to or below n based on discrete-event simulation.

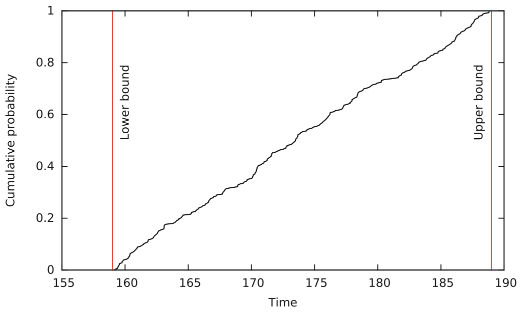
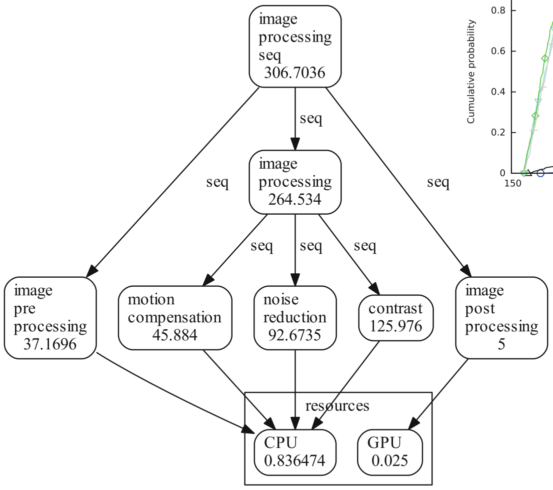
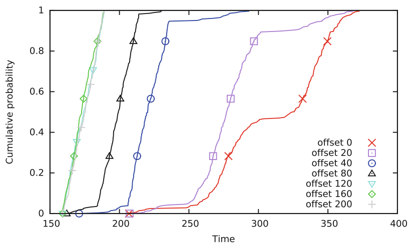


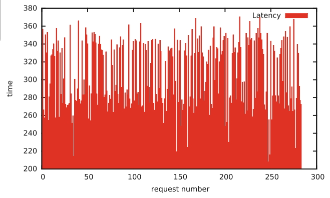
Fig. 10. The absolute lower and upper bound



(a) A latency breakdown chart (offset=0)



(b) A cumulative distribution graph



(c) A latency bar graph (offset=0)

Fig. 11. Three ways of representing latencies, generated from the iDSL code

5.2 Experiment II: Retrieving Exact Latency Distributions

Experiment II concerns generating *exact* latency distributions; the results generated using the model have to match the true values. This is accomplished by applying exhaustive methods based on model checking. Consequently, the model must be simple enough to deal with much complexity.

iDSL Model. The system (see Table 12) consists of two similar image processing services, a so-called frontal and lateral one. They are built up of the same process (see Table 10) and resource (see Table 11). The process encompasses successive high-level tasks "Noise_reduction" and "Refinement", which decompose into atomic tasks with EDF loads resulting from measurements. Notably, subprocess "Refinement" contains a nondeterministic choice, viz., atomic task "Refine" is executed either once or twice, which depends on the number of monitors connected to the iXR system. The resource (see Table 11) contains atomic resource "CPU" with rate 1 and buffersize 10 for FIFO scheduling, to which all atomic tasks are mapped. In the scenario (of Table 13), both frontal and lateral image processing are called with fixed inter-arrival times 40000. The offset of frontal is 0, the one of lateral depends on the "offset" dimension in the study (see Table 14).

Table 10. The code of an iDSL process with abstract loads

```
Section Process
  ProcessModel Image_Processing seq {
    seq Noise_reduction {
      atom Pre_processing load EDF from file "pproc"
      atom Decompose load EDF from file "dcomp"
      atom Spatial_noise_red load EDF from file "snr"
      atom Temporal_noise_red load EDF from file "tnr"
      atom Compose load EDF from file "comp" }
    seq Refinement { alt {
      atom Refine load EDF from file "ref"
        seq { atom Refine load EDF from file "ref"
      atom Refine load EDF from file "ref" } } } }
```

Table 11. The code of an iDSL resource

```
Section Resource
  ResourceModel Image_PC decomp { atom CPU rate 1 buffersize 10}
```

Table 12. The code of an iDSL system comprising two services

```
Section System
  Service Frontal_Image_Processing_Service
    Process Image_Processing
    Resource Image_PC
    Mapping assign  (a11,CPU)  scheduling policy  (CPU, FIFO)
  Service Latera1_Image_Processing_Service
    Process Image_Processing
    Resource Image_PC
    Mapping assign  (all,CPU)  scheduling policy  (CPU, FIFO)
```

Table 13. The code of an iDSL scenario with two concurrent services

```
Section Scenario
  Scenario BiPlane_Image_Processing_run
    ServiceRequest FPontal_Image_Pnocessing_Serice
      at time 0, 40000, ...
    ServiceRequest Lateral_Image_Processing_Senvice
      at time dspace("offset"), 40000+dspace("offset"), ...
```

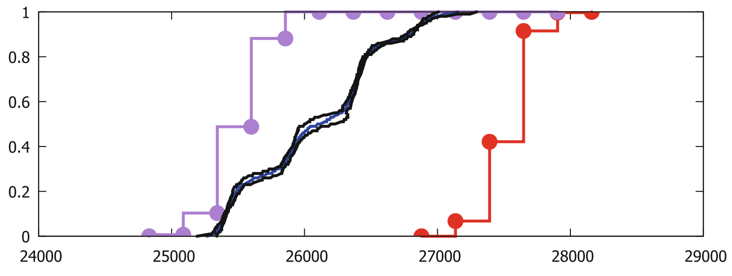
Table 14. The code of an iDSL study

```
Section Study
  Scenario BiPlane_Image_Processing_run
    DesignSpace (offset {"0" "10000" "20000" "30000"} )
```

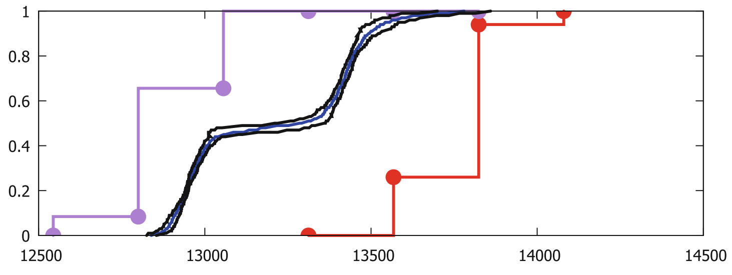
Table 15. The code of an iDSL measure

```
Section Measure
  Measure ServiceResponseTimes PTA scalable
  Measure ServiceResponseTimes PTA scalable efficient
    ServiceResponseTimes using 1 run of 1000 requests
```

Transformation. The Modest code comprises a parallel process at its highest level, with services "Frontal_Image_Processing_Service" and a lateral equivalent, a resource "CPU", a resource queue "CPU_queue", and two generators that each call a different service. The two services alternately wait for incoming requests from different generators and call the same process. Atomic processes all call "CPU_queue", providing taskloads that Resource "CPU" processes.



(a) service = Frontal IP, offset = 0



(b) service = Frontal IP, offset = 20000

Fig. 12. The lower (in purple) and upper bound CDF (in red), the simulation average (in blue), and $\alpha = 0.95$ confidence interval (in black) for two designs. (Color figure online)

Evaluation. The iSDL measure (of Table 15) encompasses two scalable Probabilistic Timed Automata measures, viz., an inefficient and efficient variant. The model is simplified using 256 cluster segments and time unit 4 (see Sect. 4.2). Additionally, a discrete-event simulation measure is added for validation.

Results. We compare the efficient approach with the inefficient one. The execution of basic estimates and discrete-event simulation takes only 27 s (<2%), but yields fairly tight bounds for model checking (as graphically depicted in Fig. 9), viz., $[47 : 99]$, $[110 : \infty]$, $[97 : 110]$ and $[97 : 110]$. Hence, in return for little time, many expensive PTA-based model checking calls can be saved. Averagely, a TA-based model checking call takes 28 s and a PTA-based one 44 s. TA-based model checking is thus useful for the binary lower & upper bound search. Overall, the efficient approach takes (for offset = 0) 1589 s (50 calls), opposed to 1937 s (61 calls) for the inefficient one.

Validation is successful; the simulation confidence intervals (in black) are located between the lower (in purple) and upper bound (in red) in Fig. 12 for two offsets, despite the application of model simplifications (of Sect. 4.2).

6 Conclusion

This paper presents a method for performance evaluation of service-oriented systems which has been put into practice using two experiments, as follows.

The Performance Evaluation Process. To gain insight in the performance of embedded systems, we have proposed a framework for performance evaluation of service-oriented systems: A *high-level performance model* is obtained by modeling the performance of a system. Optionally, this model is simplified to make it scalable, after which it is transformed into a *underlying performance model* that adheres to a widespread formalism, e.g., Stochastic Timed Automata (STA, [2,7]). Applying *performance evaluation* yields *performance results.*

Two experiments have been conducted which exemplified a performance evaluation approach that: (i) provides a domain specific, high-level modeling language; (ii) allows for the automatic evaluation of a large number of complex designs; (iii) supports different ways of performance evaluation; and, (iv) presents its results intuitively via visualizations.

References

1. Bogdoll, J., David, A., Hartmanns, A., Hermanns, H.: MCTAU: bridging the gap between Modest and UPPAAL. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 227–233. Springer, Heidelberg (2012). doi:10.1007/978-3-642-31759-0_16
2. Bohnenkamp, H., D'Argenio, P.R., Hermanns, H., Katoen, J.-P.: Modest: a compositional modeling formalism for hard and softly timed systems. IEEE Trans. Softw. Eng. **32**(10), 812–830 (2006)

3. Brinksma, H., Katoen, J.-P., Langerak, R., Latella, D.: Partial order models for quantitative extensions of LOTOS. Comput. Netw. ISDN Syst. **30**(9), 925–950 (1998)
4. D'Argenio, P.R., Hartmanns, A., Legay, A., Sedwards, S.: Statistical approximation of optimal schedulers for probabilistic timed automata. In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 99–114. Springer, Cham (2016). doi:10.1007/978-3-319-33693-0_7
5. de Gooijer, T., Jansen, A., Koziolek, H., Koziolek, A.: An industrial case study of performance and cost design space exploration. In: Proceedings of the 3rd International Conference on Performance Engineering, pp. 205–216. WOSP/SIPEW, ACM (2012)
6. Ellson, J., Gansner, E., Koutsofios, L., North, S.C., Woodhull, G.: Graphviz—open source graph drawing tools. In: Mutzel, P., Jünger, M., Leipert, S. (eds.) GD 2001. LNCS, vol. 2265, pp. 483–484. Springer, Heidelberg (2002). doi:10.1007/3-540-45848-4_57
7. Hahn, E., Hartmanns, A., Hermanns, H.: Reachability and reward checking for stochastic timed automata. Electron. Commun. Eur. Assoc. Softw. Sci. Technol. **70**, 125–140 (2014)
8. Hahn, E., Hartmanns, A., Hermanns, H., Katoen, J.-P.: A compositional modelling and analysis framework for stochastic hybrid systems. Formal Methods Syst. Des. **43**(2), 191–232 (2012)
9. Hartmanns, A.: Model-checking and simulation for stochastic timed systems. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 372–391. Springer, Heidelberg (2011). doi:10.1007/978-3-642-25271-6_20
10. Hartmanns, A., Hermanns, H.: The Modest toolset: an integrated environment for quantitative modelling and verification. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 593–598. Springer, Heidelberg (2014). doi:10.1007/978-3-642-54862-8_51
11. Haveman, S., Bonnema, M.: Requirements for high level models supporting design space exploration in model-based systems engineering. In: Procedia Computer Science, vol. 16, pp. 293–302. Elsevier (2013)
12. Johnson, J.: Designing with the Mind in Mind: Simple Guide to Understanding User Interface Design Rules. Elsevier, Amsterdam (2010)
13. Kanungo, T., Mount, D., Netanyahu, N., Piatko, C., Silverman, R., Wu, A.: An efficient K-means clustering algorithm: analysis and implementation. IEEE Trans. Pattern Anal. Mach. Intell. **24**(7), 881–892 (2002)
14. Kienhuis, B., Deprettere, E.F., van der Wolf, P., Vissers, K.: A methodology to design programmable embedded systems. In: Deprettere, E.F., Teich, J., Vassiliadis, S. (eds.) SAMOS 2001. LNCS, vol. 2268, pp. 18–37. Springer, Heidelberg (2002). doi:10.1007/3-540-45874-3_2
15. Kontogiannis, K., Lewis, G., Smith, D., Litoiu, M., Muller, H., Schuster, S., Stroulia, E.: The landscape of service-oriented systems: a research perspective. In: Proceedings of the International Workshop on Systems Development in SOA Environments. IEEE Computer Society (2007)
16. Lee, I., Leung, J.Y., Son, S.H.: Handbook of Real-Time and Embedded Systems. CRC Press, Boca Raton (2007)
17. Philippou, A., Georghiou, C., Philippou, G.: A generalized geometric distribution and some of its properties. Stat. Probab. Lett. **1**(4), 171–175 (1983)
18. Pimentel, A., Hertzberger, L., Lieverse, P., van der Wolf, P., Deprettere, E.: Exploring embedded-systems architectures with Artemis. IEEE Comput. **34**(11), 57–63 (2001)

19. Prince, J., Links, J.: Medical Imaging Signals and Systems. Pearson Prentice Hall, Upper Saddle River (2006)
20. Racine, J.: GNUplot 4.0: a portable interactive plotting utility. J. Appl. Econom. **21**(1), 133–141 (2006)
21. Rosso, C.D.: Software performance tuning of software product family architectures: two case studies in the real-time embedded systems domain. J. Syst. Softw. **81**(1), 1–19 (2008)
22. van den Berg, F.: Automated performance evaluation of service-oriented systems. Ph.D. thesis, University of Twente (2017)
23. van den Berg, F., Haverkort, B.R., Hooman, J.: Efficiently computing latency distributions by combined performance evaluation techniques. In: Proceedings of the 9th EAI International Conference on Performance Evaluation Methodologies and Tools, VALUETOOLS 2015, pp. 158–163. ICST (2015)
24. van den Berg, F., Hooman, J., Hartmanns, A., Haverkort, B.R., Remke, A.: Computing response time distributions using iterative probabilistic model checking. In: Beltrán, M., Knottenbelt, W., Bradley, J. (eds.) EPEW 2015. LNCS, vol. 9272, pp. 208–224. Springer, Cham (2015). doi:10.1007/978-3-319-23267-6_14
25. van den Berg, F., Remke, A., Haverkort, B.R.: A domain specific language for performance evaluation of medical imaging systems. In: 5th Workshop on Medical Cyber-Physical Systems. OpenAccess Series in Informatics, vol. 36, pp. 80–93. Schloss Dagstuhl (2014)
26. van den Berg, F., Remke, A., Haverkort, B.R.: iDSL: automated performance prediction and analysis of medical imaging systems. In: Beltrán, M., Knottenbelt, W., Bradley, J. (eds.) EPEW 2015. LNCS, vol. 9272, pp. 227–242. Springer, Cham (2015). doi:10.1007/978-3-319-23267-6_15
27. Zurawski, R.: Embedded Systems Handbook. CRC Press, Boca Raton (2005)