Computing Response Time Distributions Using Iterative Probabilistic Model Checking

Freek van den Berg[1](✉), Jozef Hooman[2], Arnd Hartmanns[3],
Boudewijn R. Haverkort[1], and Anne Remke[4]

[1] University of Twente, Enschede, The Netherlands
f.g.b.vandenberg@utwente.nl
[2] Radboud University, Nijmegen & TNO-ESI, Eindhoven, The Netherlands
[3] Saarland University, Saarbrücken, Germany
[4] Westfälische Wihlhems-Universität, Münster, Germany

Abstract. System designers need to have insight in the response times of service systems to see if they meet performance requirements. We present a high-level evaluation technique to obtain the distribution of services completion times. It is based on a high-level domain-specific language that hides the underlying technicalities from the system designer. Under the hood, probabilistic real-time model checking technology is used iteratively to obtain precise bounds and probabilities. This allows reasoning about nondeterministic, probabilistic and real-time aspects in a single evaluation. To reduce the state spaces for analysis, we use two sampling methods (for measurements) that simplify the system model: (i) applying an abstraction on time by increasing the length of a (discrete) model time unit, and (ii) computing only absolute bounds by replacing probabilistic choices with non-deterministic ones. We use an industrial case on image processing of an interventional X-ray system to illustrate our approach.

1 Introduction

Service-oriented systems are designed for interconnection with other systems and are commonplace in the domains of business, engineering and operations [12]. Their complexity lies in their capability to handle many service requests in parallel, for multiple kinds of services. To that end, they are equipped with multiple resources to process services requests, with variable execution times. Service-oriented systems operate in a real-time manner. When used to perform safety-critical tasks, they have to respect real-time requirements like bounded response times. In this way, their safety is determined by their performance.

Performance prediction early in design is difficult [8], especially when the system of concern does not exist yet. Simulation can provide an indication of average response times, but simulation results tend to be too optimistic. Worst-case execution time analysis [16], on the other hand, leads to absolute bounds. While

This research was supported as part of the Dutch national program COMMIT, and carried out as part of the **Allegio** project.

M. Beltrán et al. (Eds.): EPEW 2015, LNCS 9272, pp. 208–224, 2015.
DOI: 10.1007/978-3-319-23267-6_14

it can prove that *hard real-time* requirements are met, the computed bounds are often pessimistic, leading to costly, over-dimensioned implementations. For many applications, *soft real-time* guarantees are sufficient, i.e., it is acceptable if deadlines are met with a certain (high) *probability.* In such a scenario, a designer may want to know, e.g., the latency value for which, in the long run, 85 % of the service requests complete. These questions can be answered by probabilistic real-time model checking, in which probabilistic, nondeterministic and timed aspects are combined in one model.

This paper presents an approach that allows computing response time distributions using iterative probabilistic model checking. First a model is specified in a high-level domain specific language, iDSL [2], that abstracts from various under-the-hood technicalities, making it usable for systems engineers. From this model input for the Modest Toolset [7] is automatically generated in the Modest modelling language [6], which can be used for both simulations and model checking. By calling the model checking procedures iteratively, we are able to efficiently compute response time distributions in an automated fashion, which allows to better compare different designs. This is also the main difference to our previous work [2–4], where this performance evaluation trajectory has first been proposed, however without the ability to compute response time distributions precisely.

We illustrate our approach with a case study on interventional X-ray (iXR) systems as built by our industrial partner Philips Healthcare. These systems provide a continuous stream of X-ray images to a surgeon that operates on a patient. Low latency is necessary for hand-eye coordination [10], i.e., the surgeon must perceive the image stream to be real-time. Low response times of images are thus of vital importance, but a few misses of response deadlines are acceptable.

Related work. The tagged customer approach [5] is a numerical method to compute the response time distribution for open queuing networks, represented as continuous-time Markov chains (CTMCs). It may be used as a fast but approximate measure besides simulation, especially when utilizations are low and service times have high variances. The hierarchical performance evaluation tool (HIT, [1]) supports the model-based evaluation of computing system performance. HIT models are highly structured, based on functional hierarchies and modularization, as with the Y-chart philosophy [11]. HIT models are analysed using various techniques. However, the HIT model at hand determines which techniques can be used, with simulation covering the greatest spectrum. It does not have specific support for response time distributions. Modular Performance Analysis with Real-Time Calculus (MPA, [17]) is based on the Network Calculus and computes hard lower and upper bounds using event streams. Hence, these approaches do not deliver what we do.

Context. The measures that we compute and the type of systems our approach is designed for make it fall right into the field of *performance evaluation* [9]. Typical performance evaluation approaches build on fully stochastic formalisms, such as continuous-time Markov chains or stochastic Petri nets. However, our examples require a mixture of deterministic timing with probabilistic effects and concurrency; we want also to be able to compute hard bounds; and we use abstraction techniques for model simplification that introduce nondeterministic delays. Since they capture exactly these aspects in a compositional fashion, we chose probabilistic timed automata (PTA, [14]) as the semantic basis of our models. The analysis of PTA is supported by a number of tools including Prism [13] and the Modest Toolset [7]. We use the latter due to its high-level input language and the ability to perform model checking using the included mcsta tool as well as simulation using the modes simulator.

2 Problem Statement and Case Study

We describe service-oriented systems (Section 2.1) and their performance characteristics (Section 2.2), and use the case study on iXR systems as an example.

2.1 Service-Oriented Systems

In service-oriented systems, the system receives a service request after which the system replies with a service response that completes the request. The latency is the elapsed time between service request and response. A service decomposes into a number of atomic tasks that each require access to resources (e.g. a CPU, I/O bus or GPU) for computation or data transfer. These tasks may have variable execution times. A service system can make use of multiple instances of one or more services at the same time, which gives rise to concurrency among service instances. A scheduling policy resolves this concurrency by prescribing an order in which tasks gain access to resources, e.g., first-in, first-out (FIFO). A scenario describes when these service requests arrive; for example, "a service system receives one service request every $100ms$, forever". Service systems can have one or more configurations, each with their own properties.

Example: iXR Systems. provide a continuous stream of images to support a surgeon that operates a patient, i.e., they provide an image processing (IP) service. Service requests are incoming, unprocessed images that arrive with fixed inter-arrival times. The iXR system responds with processed images. The latency should be low enough to enable hand/eye-coordination [10]. The system comprises one resource, the CPU. The service decomposes into a pipeline of twelve image processing steps, all performed on the CPU via a FIFO scheduling policy. Service IP receives 10 images per second. We consider two configurations having image resolutions of 512^2 and 1024^2 pixels, respectively.

2.2 Performance Questions

We define performance questions of service-oriented systems to assess their performance. There are black-box and white-box measures. Black-box measures are

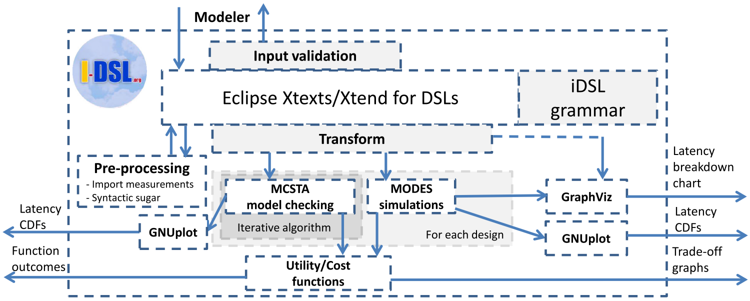


Fig. 1. The iDSL solution chain

observable from the outside of the system, and examples include the latency per service. White-box measures, such as resource utilizations, require knowledge about the inside of the system. We focus on service latencies because they are the prime measure of interest in the case study on iXR systems, viz., latencies should be low enough to enable hand/eye coordination. We show two categories of performance questions that are concerned with service requests of a given service in a given service system, operating in a given scenario, as follows:

Q1. What is the latency for which a given percentage of the service requests completes?

Q2. Which percentage of the service requests has a latency below a given value?

Example: iXR systems have performance questions that are as follows:
Q1a. What is the latency for which 85% of the service requests completes?
Q2a. Which percentage of the service requests has a latency below 55ms?

3 A Formal Model for Service Systems

iDSL comprises a high-level language for modelling service systems, and a toolset to evaluate their performance (see Figure 1 for its solution chain). Each iDSL model leads to the generation of performance artefacts for many so-called designs. iDSL has been developed using the Xtext and Xtend plug-ins of Eclipse for Domain Specific Languages (DSLs). iDSL is thus an Eclipse plug-in with an extensive Integrated development environment (IDE). In a pre-processing phase, measurements can be imported into the model and syntactic sugar is resolved. For each design, performance analysis is done via multiple mcsta (see Section 4) and modes calls of the Modest Toolset [7] for model checking and simulation, respectively. Visualizations are generated with Graphviz and GNUplot.

In Section 3.1, we provide the syntax of the iDSL language by showing its key language constructs. We apply it to iXR systems in Section 3.2 and show three sampling methods for measurements in Section 3.3. In Section 3.4, we define utility and cost functions that answer performance questions using a query on the computed results. In Section 3.5, we define the semantics of iDSL by describing the transformation from iDSL to Modest.

3.1 iDSL Language Syntax

We specify service systems formally using iDSL [2], following the six concepts as illustrated in Figure 2

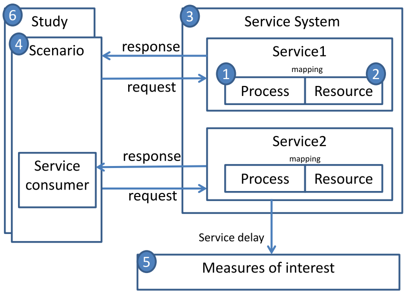
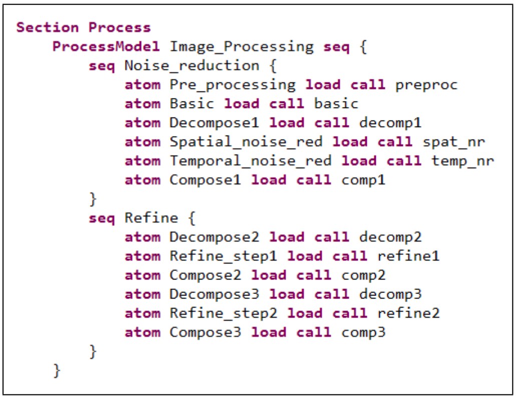


Fig. 2. The iDSL language's concepts.

1. A *process* decomposes service requests into atomic tasks. iDSL provides the following process algebra constructs: *palt*, a probabilistic choice among alternatives; *alt*, a non-deterministic choice between alternatives; *par* for parallel activities; and *seq* for sequential activities. iDSL also offers a *mutex*, a mutual exclusion to run processes uninterruptedly.
2. *Resources* are capable of performing one atomic task at a time, in a certain amount of time. A *mapping* assigns atomic tasks to resources.
3. A *service system* consists of one or more services, each implemented using a process, a set of resources and a mapping between processes and resources.
4. A *scenario* comprises a number of invoked service requests over time to observe the performance behaviour of the system in specific circumstances.
5. *Measures of interest* define which performance measures are obtained.
6. A *study* evaluates a selection of systematically chosen systems and scenarios.

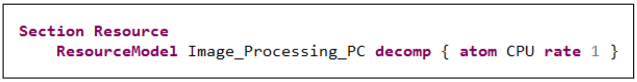
3.2 iDSL Model of iXR Systems

Let us now explain the iDSL model of an iXR system:



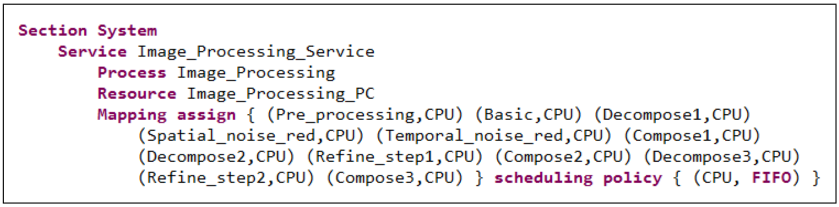
```
Section Process
    ProcessModel Image_Processing seq {
        seq Noise_reduction {
            atom Pre_processing load call preproc
            atom Basic load call basic
            atom Decompose1 load call decomp1
            atom Spatial_noise_red load call spat_nr
            atom Temporal_noise_red load call temp_nr
            atom Compose1 load call comp1
        }
        seq Refine {
            atom Decompose2 load call decomp2
            atom Refine_step1 load call refine1
            atom Compose2 load call comp2
            atom Decompose3 load call decomp3
            atom Refine_step2 load call refine2
            atom Compose3 load call comp3
        }
    }
```

1. Section **process** (right) contains process "Image_-processing" that specifies how images are processed, viz., via two high-level operations "Noise_reduction" and "Refine". They in turn decompose in a sequential pipeline of twelve image operations, each with a load (an amount of work), specified via an abstraction mechanism. These loads are assumed to be independent. Section 3.3 shows how these abstract loads are implemented.
2. Section **resource** comprises resource "Image_processing_PC", that has a CPU with a rate of 1, i.e., it can process 1 unit of load per μs, the time unit of choice. The resource is defined as follows.



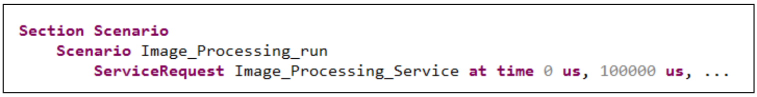
```
Section Resource
    ResourceModel Image_Processing_PC decomp { atom CPU rate 1 }
```

3. Section **system** comprises a service named "Image_Processing_Service", which connects process "Image_Processing" to resource "Image_Processing_PC" by defining a mapping, i.e., each of the twelve image operations is performed on the CPU, and a FIFO scheduling policy is used to resolve concurrency.



```
Section System
    Service Image_Processing_Service
        Process Image_Processing
        Resource Image_Processing_PC
        Mapping assign { (Pre_processing,CPU) (Basic,CPU) (Decompose1,CPU)
            (Spatial_noise_red,CPU) (Temporal_noise_red,CPU) (Compose1,CPU)
            (Decompose2,CPU) (Refine_step1,CPU) (Compose2,CPU) (Decompose3,CPU)
            (Refine_step2,CPU) (Compose3,CPU) } scheduling policy { (CPU, FIFO) }
```

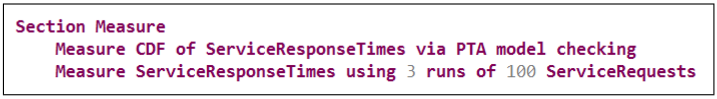
4. Section **scenario** comprises scenario "Image_Processing_Run", in which the service is invoked 10 times per second, i.e., once every 100000 s, forever.



```
Section Scenario
    Scenario Image_Processing_run
        ServiceRequest Image_Processing_Service at time 0 us, 100000 us, ...
```

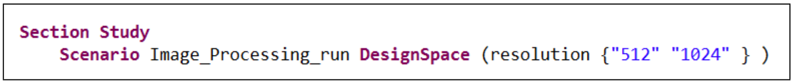
5. Below, section **measure** contains two measures: "ServiceResonseTimes" retrieves average latencies of 100 service requests via simulations, using 3 runs. Simulations provide quick insight into the general behaviour of a system, but are less suitable for showing the extreme behaviour of a system.

Measure "CDF ..." yields a cumulative distribution function (CDF) with latencies, obtained via probabilistic model checking. As usual, a CDF is a function that displays for each latency value l, the percentage of the service requests that has a latency below l, e.g., $cdf(60) = 0.5$ means that half of all service requests have a latency below 60 ms. This measure is obtained via model checking and is thus much slower than simulation, but conveys different insights, e.g., absolute lower and upper bounds. It is explained in detail in Section 4.



```
Section Measure
    Measure CDF of ServiceResponseTimes via PTA model checking
    Measure ServiceResponseTimes using 3 runs of 100 ServiceRequests
```

6. Finally, section **study** allows for design instances to be defined that are each evaluated using the defined measures. We model two iXR systems having image resolutions 512^2 and 1024^2 pixels, respectively.



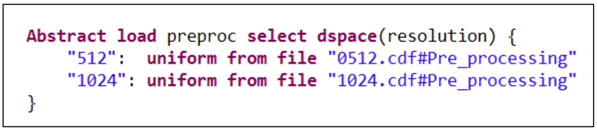
```
Section Study
    Scenario Image_Processing_run DesignSpace (resolution {"512" "1024" } )
```

iDSL offers two approaches to handle uncertainty: probabilism and nondeterminism. *Probabilism* specifies a range of weighted outcomes, e.g., an image operation completes with probability 0.6 in 45 s, and 0.4 in 46 s. Non-determinism is similar but without probabilities, e.g., an image operation completes in either 45 s, 46 s, or between 45 s and 46 s. *Nondeterminism* can also occur when a system processes multiple service requests, i.e., if a resource is potentially accessed multiple times at the same time, the order of action is undefined.

3.3 Three Sampling Methods for Measurements

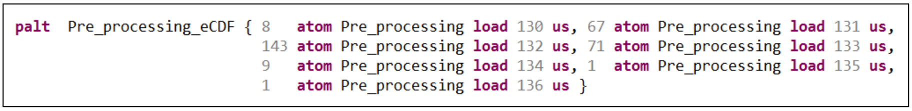
In Section 3.2 (process), twelve image operations are defined using abstract loads. In our case, these loads are based on 300 latency measurements each, performed on a real iXR system, to calibrate the model. We show three implementations for "pre-processing", each corresponding to a different way of sampling, viz., *uniform*, *abstract time*, and *non-deterministic time* sampling. The other image operations are implemented similarly.

Uniform Sampling. Below, the abstract load "preproc" defines the variable load of "Pre_processing", using uniform sampling. In uniform sampling, each measurement has an equal probability to be sampled. The load is defined for two image resolutions, viz., 512^2 and 1024^2 pixels. The *dspace* operator selects the right set of measurements, depending on the resolution of the design instance at hand. *Uniform from file* refers to an external file and a position in that file at which the correct measurements are stored.



```
Abstract load preproc select dspace(resolution) {
    "512":  uniform from file "0512.cdf#Pre_processing"
    "1024": uniform from file "1024.cdf#Pre_processing"
}
```

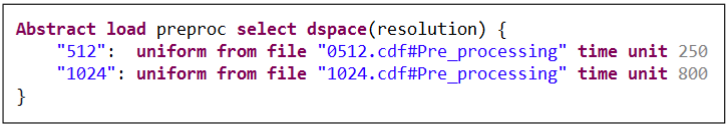
iDSL transforms the implementations into basic process algebra constructs, via a so-called model transformation. This leads to one palt-construct per resolution, consisting of measurements. For resolution 512^2, it is as follows:



```
palt  Pre_processing_eCDF { 8   atom Pre_processing load 130 us, 67 atom Pre_processing load 131 us,
                            143 atom Pre_processing load 132 us, 71 atom Pre_processing load 133 us,
                            9   atom Pre_processing load 134 us, 1  atom Pre_processing load 135 us,
                            1   atom Pre_processing load 136 us }
```

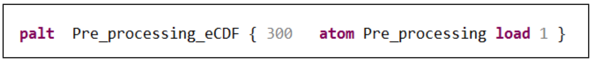
E.g., the probability for 130 s is $\frac{8}{300}$, because 8 out of 300 measurements are 130 s, for 131 s it is $\frac{67}{300}$, because 67 out of 300 measurements are 131 s, etc.

Abstract Time Sampling. Next, the abstract load "preproc" defines the variable load of "Pre_processing" using abstract time sampling, as follows.



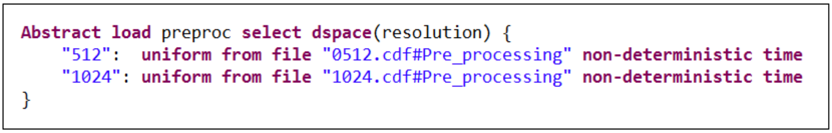
```
Abstract load preproc select dspace(resolution) {
    "512":  uniform from file "0512.cdf#Pre_processing" time unit 250
    "1024": uniform from file "1024.cdf#Pre_processing" time unit 800
}
```

In abstract time sampling, measurements are divided by a given constant number and rounded to make the model simpler at the price of some precision. We divide by 250 for resolution 512^2, and 800 for 1024^2. The final results will be multiplied by the same constants again. The result of the model transformation for resolution 512^2, is as follows.



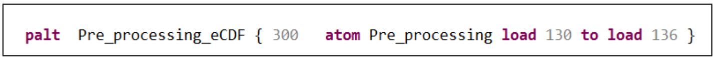
```
palt  Pre_processing_eCDF { 300   atom Pre_processing load 1 }
```

Non-deterministic Time Sampling. Finally, "preproc" defines the variable load of "Pre_processing" using non-deterministic time sampling. It is as follows:



```
Abstract load preproc select dspace(resolution) {
    "512":  uniform from file "0512.cdf#Pre_processing" non-deterministic time
    "1024": uniform from file "1024.cdf#Pre_processing" non-deterministic time
}
```

In non-deterministic time sampling, the time of an image operation is defined as the smallest segment that contains all measurements, as follows.



```
palt  Pre_processing_eCDF { 300   atom Pre_processing load 130 to load 136 }
```

Non-deterministic time sampling is typically used to obtain absolute latency bounds. Semantically, the above means that any real value in segment $[130 : 136]$ is a valid sample, but that their individual probabilities are unknown.

3.4 Performance Queries in iDSL

In the following, we add performance queries to the iDSL model. They are specified as so-called *utility* and *cost* functions that specify a query on the performance results, and return a real number. They rely on measures. In the case study, we show two measures that are based on simulations and model checking, resp.

A function is either a cost function when lower values are preferred, e.g., the average latency of a service, or a utility function when higher values are preferred, e.g., the percentage of service requests completed after some time.

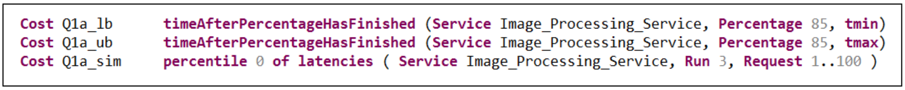
The iDSL model comprises both the system model and a scenario in which it operates. We analyze the response times to service requests of a given service S of this system, the latencies. To this end, we introduce four model checking-based functions and two simulation functions, based on Q1 and Q2 (see Section 2.2).

First, we introduce two pairs of model checking functions.
1. Function $Q1a_{lb}$ ($Q1a_{ub}$) returns the minimum (maximum) latency before which P percent of the service requests of service S complete.
2. Function $Q2a_{lb}$ ($Q2a_{ub}$) returns the minimum (maximum) percentage of service requests of service S that has a latency below time T.
Second, we introduce two similar simulation based functions:
1. Function $Q1a_{sim}$ returns the latency before which P percent of the service requests of service S complete, based on R simulation runs of Rq requests each.
2. Function $Q2a_{sim}$ returns the percentage of service requests of service S that has a latency below time T, based on R simulation runs of Rq requests each.

Note that model checking-based functions have two variants, viz., a minimum and maximum one; they return bounds. Simulation has two parameters: runs and requests. The higher these values are, the more accurate the results will be.

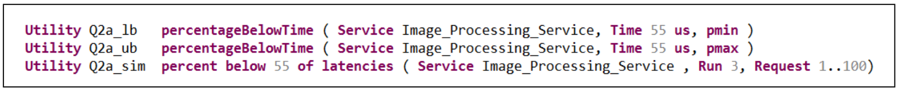
Example: iXR Systems. We define two groups of performance queries for iXR systems in iDSL, which are added to the Measure section of iDSL. Each performance question is defined three times, viz., twice for model checking and once for simulation. Simulations are based on 3 runs of 100 requests each.

Questions of type Q1 ask latencies before which a given percentage of service requests completes. They are cost functions, since lower latencies are preferred:



```
Cost Q1a_lb     timeAfterPercentageHasFinished (Service Image_Processing_Service, Percentage 85, tmin)
Cost Q1a_ub     timeAfterPercentageHasFinished (Service Image_Processing_Service, Percentage 85, tmax)
Cost Q1a_sim    percentile 0 of latencies ( Service Image_Processing_Service, Run 3, Request 1..100 )
```

Questions of type Q2 ask percentages of service requests that have latencies below a given time. They are utility functions since higher values are preferred:



```
Utility Q2a_lb   percentageBelowTime ( Service Image_Processing_Service, Time 55 us, pmin )
Utility Q2a_ub   percentageBelowTime ( Service Image_Processing_Service, Time 55 us, pmax )
Utility Q2a_sim  percent below 55 of latencies ( Service Image_Processing_Service , Run 3, Request 1..100)
```

3.5 Translation to Modest

The semantics of the iDSL language is specified via a transformation from iDSL models to Modest models. Modest is a high-level modelling language rooted in process algebra with a formal semantics in terms of stochastic hybrid automata (SHA) [6]. Several other popular formalisms such as PTA and discrete-time Markov chains are special cases of SHA. The analysis of Modest models is supported by the Modest Toolset [7], which in particular includes the tools modes for simulation (or: statistical model checking) and mcsta for model checking of Modest models conforming to the PTA subset of the language.

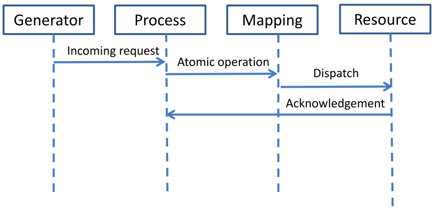


Fig. 3. An interaction diagram.

iDSL is a high-level language specifically tailored to service systems, yielding, for the iXR case, a model that is seven times smaller textually than the autogenerated Modest code. Additionally, small architectural changes to the iDSL model can affect the whole Modest code, making a Modest-only approach hard w.r.t. maintenance.

An iDSL model transforms into one or more Modest models, each containing an overarching Modest process. This process decomposes in a number of parallel sub-processes of class *generator*, *process*, *mapping* or *resource*. They interact in the way as shown in Figure 3, viz., a *generator* triggers a *process*, which in turn, via a mapping, obtains access to a *resource* and receives an acknowledgement.

iDSL processes transform into processes of class *process* in Modest. Process algebra constructs in iDSL are thereby translated to their Modest counterparts. Processes also contain calls to *mapping* processes, for each atomic task. iDSL resources become *resource* processes in Modest, containing a queue and a counter for service time. Mappings lead to *mapping* processes that each connect an atomic tasks to a resource. iDSL systems and services do not lead to Modest code, but merely organize the iDSL model. In the iDSL scenario, service requests lead to a Modest generator that sends a trigger to a process periodically and forever. Measures lead to one or more Modest models, depending on the measure. In the next section, we show how multiple Modest models are generated for model checking purposes. An iDSL study contains design instances that are evaluated separately. Separate Modest models are generated for each design.

4 Computing Response Time Distributions

We present a new approach to compute latency response times, based on iterative probabilistic model checking. It answers performance questions of iDSL models (of Section 3.4) in five steps: (i) iDSL models are transformed into Modest models that are used to retrieve service latencies (Sect. 4.1); (ii) latencies are aggregated into one overarching latency per service (Sect. 4.2); (iii) mcsta is applied iteratively to obtain probability bounds (Sect. 4.5); (iv) these bounds are transformed into a set of possible CDFs (Sect. 4.4); and (v) performance questions are answered using the set of possible CDFs (Sect. 4.5).

4.1 From iDSL Queries to Modest

We generate a range of Modest models to answer performance queries, for each iDSL model i, each service s within that model, and both the minimum and maximum probability (a flag f). The models have one parameter, $t \in \mathbb{R}_{\geq 0}$, and return probability p: the probability that a service completes within time t.

The Modest models are generated using the transformation in Section 3.5. Also, a measure is added to the specific service the model measures, i.e., its process is enclosed by stopwatches that record latencies of its service requests. Finally, a property to retrieve the minimum or maximum probability ($p_{\min}$ or $p_{\max}$) that a service completes within time t is added. Hence, Modest models are reused to obtain many probabilities, for many values of time t. For the sake of simplicity, we specify an abstract function $\mathcal{M}$ that retrieves such a probability:

$$p = \mathcal{M}(i, s, f, t),$$

where p is either the minimum or maximum probability (depending on flag f) that service s in iDSL model i completes within time t.

$\mathcal{M}$ is implemented using the following three steps: (i) select the Modest model of iDSL model i, service s and f; (ii) run this model in mcsta with parameter time t; and (iii) return the result of mcsta as probability p.

4.2 Aggregating Latencies of Service Requests

Modest models have been generated that return the probability that a service request completes within a given time. In iDSL, however, a service leads to an infinite stream of service requests, each with their own latency. Ideally, the average of these latencies is a measure for the performance of the whole service:

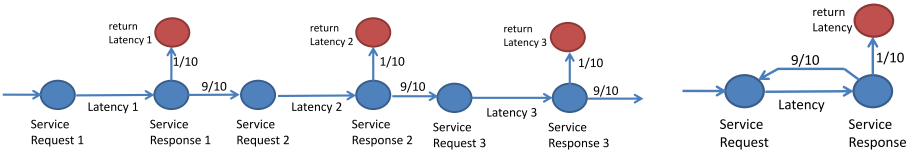
$$P_{\Omega}(t) = \lim_{k\to\infty} \frac{1}{k} \sum_{n=1}^{k} P_n(t), \tag{1}$$

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. However, this infinite sum is not computable. Hence, we show the following two weighted averages of the latencies that can be used to approximate the measure.

First, the *arithmetic mean* considers the first N service requests and weighs them equally, as follows:

$$P_{\Omega}(t) = \frac{1}{N} \sum_{n=1}^{N} P_n(t), \tag{2}$$

where $N \in \mathbb{N}^{+}$ is the number of service requests considered, e.g., $N = 100$. It is similar to (1) for large values of N. However, even for small values of N, it



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

Fig. 4. Binary probabilistic choices induce the geometric distribution

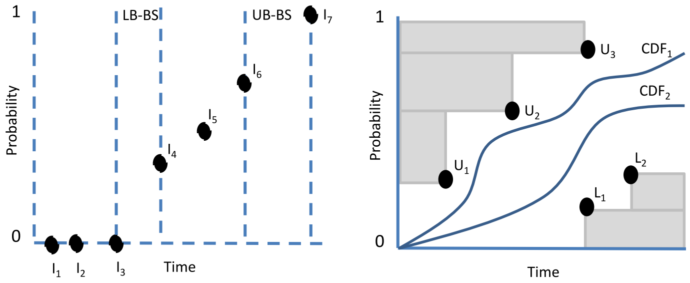
has two drawbacks: (i) it requires a counter to be added to the state in Modest to keep track of the service request number; and (ii) latencies of the $(N+1)^{th}$ service request and later are neglected.

Second, the *geometric distribution* [15] weighs service requests exponentially decreasing, as follows:

$$P_{\Omega}(t) \; = \; \sum_{n=1}^{\infty} (1-\rho)^{n-1} \, \rho \, P_n(t), \tag{3}$$

where $\rho \in (0:1)$ is the parameter of the geometric distribution.

It is, again, similar to (1) for ρ close to 0. Lower ρ-values lead to a more complex model but more precise results, and vice versa. Since the geometric distribution considers all service requests and it is capable of finding absolute maximum latencies. In Modest, it is implemented as a binary probabilistic choice every time a service request completes (as depicted in Figure 4a): either the currently measured latency is returned, with probability ρ, or the next service request is evaluated, with probability $1-\rho$. Moreover, the geometric distribution is memoryless, i.e., the binary choice does not rely on state information. Consequently, it is possible to omit the service request number from the model, leading to a single reoccurring service request (as in Figure 4b). In the remainder of this paper, we only consider the geometric distribution with $\rho = \frac{1}{10}$, empirically determined.



(a) The initial scan and resulting bounds

(b) The set of possible CDFs

Fig. 5. Cumulative Distribution Functions (CDFs) based on function $\mathcal{M}$

4.3 Iterative Model Checking for Probability Bounds

We provide an algorithm to compute function $\mathcal{M}$, for a given iDSL model i, service s in this model and a minimum/maximum bound flag f. $\mathcal{M}(i, s, f, t)$ is iteratively applied for different values t, comprising three stages, viz., an initial scan, a binary lower & upper bound search, and a brute force computation.

Initial Scan. The initial scan gives an idea of the order of magnitude of the time values. We compute $\mathcal{M}(i, s, p_{\min/\max}, t)$ for $t = 1, 2, 4, 8, 16, \ldots, 2^m, 2^{m+1}, \ldots, 2^n, 2^{n+1}$ until $\mathcal{M}(i, s, p_{\min/\max}, 2^{n+1}) = 1$. The lower bound is then located between 2^m and 2^{m+1} with $\mathcal{M}(i, s, p_{\min/\max}, 2^m) = 0$ and $\mathcal{M}(i, s, p_{\min/\max}, 2^{m+1}) > 0$, and the upper bound between 2^n and 2^{n+1} with $\mathcal{M}(i, s, p_{\min/\max}, 2^n) < 1$ and $\mathcal{M}(i, s, p_{\min/\max}, 2^{n+1}) = 1$. Note that m and n are unique values.

Figure 5a depicts the initial scan graphically. It shows computations i_1, i_2, $\ldots$, i_7, with i_7 having a probability of 1. We observe that the lower bound is located between i_3 and i_4, and the upper bound between i_6 and i_7.

Binary Lower & Upper Bound Search. Next, two binary searches are performed to determine the *exact* lower and upper bound, using the ranges of the initial scan. The binary searches are applied to $[2^m : 2^{m+1}]$ and $[2^n : 2^{n+1}]$ for the lower and upper bound, respectively. They lead to lower and upper bound lb and ub, respectively. By definition, $\mathcal{M}(i, s, p_{\min/\max}, t) = 0$, for $t < lb$, and $\mathcal{M}(i, s, p_{\min/\max}, t) = 1$, for $t > ub$. Thus, only $\mathcal{M}(i, s, p_{\min/\max}, t)$, for $t \in [lb : ub]$, need to be determined yet.

Brute Force Computation. We obtain $\mathcal{M}(i, s, p_{\min/\max}, t)$ for all times $t \in [lb : ub]$. We compute them on c CPU cores by distributing the possible values for t equally to the available CPU cores.

Finally, a cache is used throughout all computations for $\mathcal{M}$ to avoid duplicate computations, which is possible because $\mathcal{M}$ is deterministic.

4.4 Transforming Bounds into a Set of Possible CDFs

By iteratively computing values of function $\mathcal{M}$, lower and upper bound probabilities ($p_{\min}$ and $p_{\max}$) of latencies have been computed, for a given iDSL model i and service s. Figure 5b shows five probabilities (upper bounds U_1, U_2 and U_3, and lower bounds L_1 and L_2) and two CDFs that respect these bounds. We consider the set of all CDFs that respect these bounds, i.e., each CDF is below the upper bounds and above the lower bounds, for all times t. Formally, function $\mathit{CDF}_{all} \colon I \times S \to 2^{\widehat{\mathit{CDF}}}$ returns, where $\widehat{\mathit{CDF}}$ is the universe of all CDFs, given an iDSL model i and service s, the set of CDFs that respect the bounds in $\mathcal{M}$:

$$\begin{aligned}\mathit{CDF}_{all}(i, s) = \{\, \mathit{cdf} \in \widehat{\mathit{CDF}} \mid \mathit{cdf}(0) = 0 \;\wedge\; &\mathcal{M}(i, s, p_{\min}, t) = p_1 \Rightarrow \mathit{cdf}(t) \geq p_1 \\ \wedge\; &\mathcal{M}(i, s, p_{\max}, t) = p_2 \Rightarrow \mathit{cdf}(t) \leq p_2 \,\}\end{aligned}$$

Constraint $\mathit{cdf}(0)$ requires all the values to be greater than or equal to 0.

4.5 Answering the Performance Queries Using the CDFs

We now use CDF_{all} to answer the performance queries, as follows. Queries of type Q1, the minimum time for which a service request completes with probability p, are determined, as follows:

$$Q1(i, s, p, T_{\min}) = \min \{\, t \mid (t, p) \in \mathit{cdf} \wedge \mathit{cdf} \in \mathit{CDF}_{all}(i, s) \,\}$$

Queries of type Q2, the minimum probability that a latency is below a given time t, are determined, as follows:

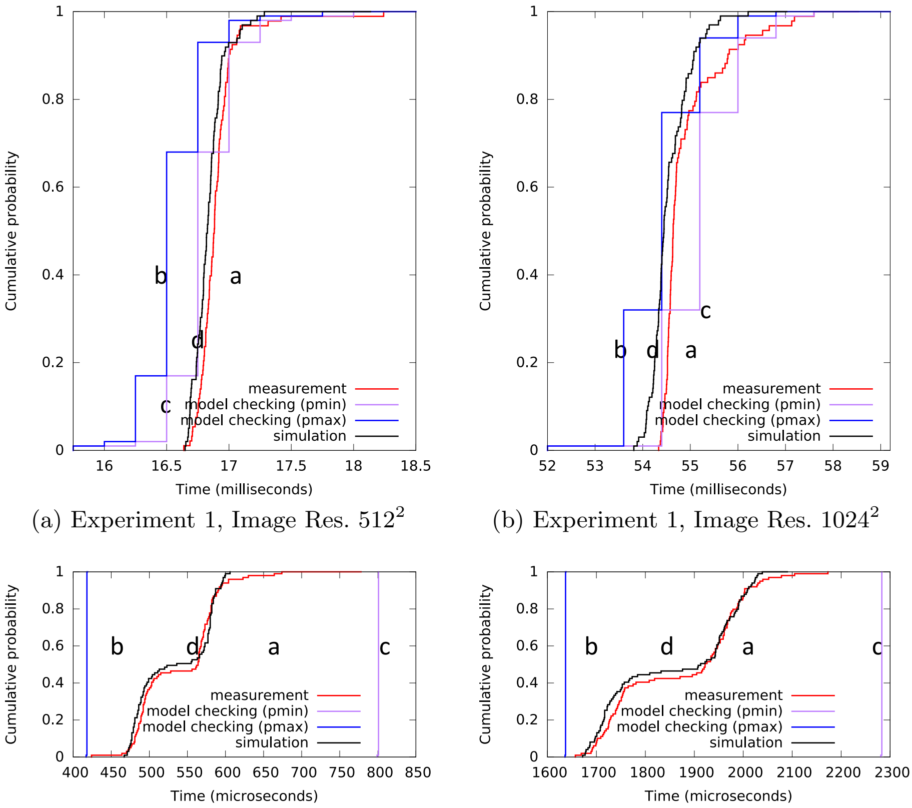
$$Q2(i, s, t, P_{\min}) = \min \{ p \mid (t, p) \in \mathit{cdf} \wedge \mathit{cdf} \in \mathit{CDF_{all}}(i, s) \}$$

The maximum cases of Q1 and Q2 are determined analogously.

5 Case Study Results

We apply the performance analysis approach of Section 4 to the iXR system of the case study. We define three experiments and compare their results with simulations and real measurements, in three steps: (i) we present CDFs with latency times; (ii) we show the execution times and model sizes; and (iii) we show the answers to the performance questions (for Experiment 1).

Three experiments are defined, based on the sampling methods in Section 3.3, respectively. Experiment 0 uses uniform sampling. Running MCSTA leads



(a) Experiment 1, Image Res. 512^2

(b) Experiment 1, Image Res. 1024^2

(c) Experiment 2, Image Res. 512^2

(d) Experiment 2, Image Res. 1024^2

Fig. 6. CDFs with latencies of IP of iXR systems: measurements on a real iXR system (a), model checking (lower & upper bounds) (b+c) and simulations (d).

to a incremental generation of the state space, but runs out of memory and stalls after having generated 38 million states. Experiment 1 uses abstract time sampling. Experiment 2 uses non-deterministic time sampling, but on a subset of the system to reduce complexity, i.e., only the first 3 image operations are considered, instead of all 12.

CDFs with Latencies. Figure 6 shows latency CDFs, for resolutions 512^2 and 1024^2, and experiment 1 and 2. They show the percentage of service requests that complete, on the Y-axis, within a given latency, on the X-axis.
Experiment 1: Figure 6a and 6b convey that the bounds, obtained by model checking, do not enclose all measurements in both cases, e.g., in Figure 6a the lower bound has probability 0.16 for time 16.5 while measurements are close to 0. This imprecision results from the time abstraction. Also, model checking returns higher time values than simulations for probabilities close to 1.
Experiment 2: Figure 6c and 6d show that the computed bounds enclose all measurements, as required. Simulations show, compared with measurements, more average behaviour, i.e., less variance and tighter bounds.

Table 1. Two experiments: execution times for simulations and model checking, the number of states of the Modest model and the number of MCSTA calls.

Exp	**sim. time**	**MC time**	**img. res.**	**states**	**calls**
Experiment 1: abstract time	56”	3:17’28”	512^2	8.05M	88
			1024^2	1.29M	85
Experiment 2: bounds only	44”	5:59’22”	512^2	2.03M	49
			1024^2	2.77M	57

Execution Times and Complexities. Table 1 shows for Experiment 1 and 2 the execution times (on an AMD A6-3400M APU, 8 GB RAM system) and state space sizes. All simulations finish within a minute, whereas model checking takes in the order of hours, i.e., up to 500 times longer. The number of states ranges from 1.29 million to 8.05 million. mcsta is called up to 88 times.

Table 2. Performance questions outcomes

		512^2			1024^2		
	n	**sim**	**lb**	**ub**	**sim**	**lb**	**ub**
Q1a	85	16.9	16.8	17.0	55.0	55.2	56.0
Q1b	0	16.6	15.8	15.8	53.8	52.0	52.0
Q1c	50	16.8	16.5	16.8	54.4	54.4	55.2
Q1d	90	17.0	16.8	17.0	55.1	55.2	56.0
Q1e	100	18.1	18.5	18.5	57.0	59.2	59.2
Q2a	55	x	x	x	84%	32%	77%
Q2b	17	91%	91%	96%	x	x	x

Results of the Performance Queries. Table 2 shows the answers to the performance queries for Experiment 1 (as obtained in Section 4.4 and 4.5).

Table 2 (top) shows that model checking leads to lower values (for $n = 0$), via comparable values (for $n = 50$), to higher values (for $n = 100$) than simulations, i.e., it has a higher variance. For resolution 1024^2, Model checking values are higher , for $n = 85$. This difference even increases for $p = 90$ and $p = 100$.

Table 2 (bottom) shows that, for a given latency, the percentage of service requests that meet a latency deadline can be obtained. E.g., if 90 % of the images need to be in time, then a latency of 17 ms for resolution 512^2 is met.

6 Conclusion

We have introduced a high-level domain specific language to model service systems and retrieve their response time distributions, usable by system designers. Besides the traditionally used simulations, response times are also obtained via iterative probabilistic model checking. Since model checking faces the state space explosion problem, we have introduced sampling methods to reduce the model complexity: (i) increasing the model time unit, and (ii) eliminating probabilism. A case studty on iXR systems shows the feasibility of our approach.

References

1. Beilner, H., Mater, J., Weissenberg, N.: Towards a performance modelling environment: news on HIT. In: Modeling Techniques and Tools for Computer Performance Evaluation, pp. 57–75. Plenum Press (1989)
2. 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, pp. 80–93. Schloss Dagstuhl (2014)
3. van den Berg, F., Remke, A., Haverkort, B.: iDSL: Automated performance prediction and analysis of medical imaging systems. In: Computer Performance Engineering, LNCS, vol. 9272. Springer (2015) (to appear)
4. van den Berg, F., Remke, A., Mooij, A., Haverkort, B.: Performance evaluation for collision prevention based on a domain specific language. In: Balsamo, M.S., Knottenbelt, W.J., Marin, A. (eds.) EPEW 2013. LNCS, vol. 8168, pp. 276–287. Springer, Heidelberg (2013)
5. Grottke, M., Apte, V., Trivedi, K., Woolet, S.: Response time distributions in networks of queues. In: Queueing Networks, pp. 587–641. Springer (2011)
6. Hahn, E., Hartmanns, A., Hermanns, H., Katoen, J.P.: A compositional modelling and analysis framework for stochastic hybrid systems. Formal Methods in System Design **43**(2), 191–232 (2012)
7. Hartmanns, A., Hermanns, H.: The modest toolset: an integrated environment for quantitative modelling and verification. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 593–598. Springer, Heidelberg (2014)
8. Haveman, S., Bonnema, G., van den Berg, F.: Early insight in systems design through modeling and simulation. Procedia Computer Science **28**, 171–178 (2014)
9. Jain, R.: The Art of Computer Systems Performance Analysis. John Wiley & Sons (1991)
10. Johnson, J.: Designing with the Mind in Mind: Simple Guide to Understanding User Interface Design Rules. Morgan Kaufmann (2010)
11. Kienhuis, B., Deprettere, E.F., van der Wolf, P., Vissers, K.: A methodology to design programmable embedded systems. In: Deprettere, F., Teich, J., Vassiliadis, S. (eds.) SAMOS 2001. LNCS, vol. 2268, pp. 18–37. Springer, Heidelberg (2002)

12. Kontogiannis, K., Lewis, G., Smith, D. and 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, p. 1. IEEE Computer Society (2007)
13. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011)
14. Kwiatkowska, M., Norman, G., Segala, R., Sproston, J.: Automatic verification of real-time systems with discrete probability distributions. Theor. Comput. Sci. **282**(1), 101–150 (2002)
15. Philippou, A., Georghiou, C., Philippou, G.: A generalized geometric distribution and some of its properties. Statistics & Probability Letters **1**(4), 171–175 (1983)
16. Puschner, P., Burns, A.: Guest editorial: A review of worst-case execution-time analysis. Real-Time Systems **18**(2–3), 115–128 (2000)
17. Wandeler, E., Thiele, L., Verhoef, M., Lieverse, P.: System architecture evaluation using modular performance analysis: a case study. International Journal on Software Tools for Technology Transfer **8**(6), 649–667 (2006)