A Domain Specific Language for Performance Evaluation of Medical Imaging Systems*

Freek van den Berg, Anne Remke, and Boudewijn R. Haverkort

Design and Analysis of Communication Systems, University of Twente
P.O. Box 217, 7500 AE, Enschede, The Netherlands
{f.g.b.vandenberg,a.k.i.remke,b.r.h.m.haverkort}@utwente.nl

Abstract

We propose iDSL, a domain specific language and toolbox for performance evaluation of Medical Imaging Systems. iDSL provides transformations to MoDeST models, which are in turn converted into UPPAAL and discrete-event MODES models. This enables automated performance evaluation by means of model checking and simulations. iDSL presents its results visually. We have tested iDSL on two example image processing systems. iDSL has successfully returned differentiated delays, resource utilizations and delay bounds. Hence, iDSL helps in evaluating and choosing between design alternatives, such as the effects of merging subsystems onto one platform or moving functionality from one platform to another.

1998 ACM Subject Classification B.8.2 Performance Analysis and Design Aids

Keywords and phrases Domain Specific Language, Performance Evaluation, Simulation, Model Checking, Medical Systems

Digital Object Identifier 10.4230/OASIcs.MCPS.2014.80

1 Introduction

Medical imaging systems (MIS) are used to perform safety critical tasks. Their malfunctioning can lead to serious injury [1]. The safety is, among others, significantly determined by their performance, since imaging applications are time critical by nature. Predicting the performance of MIS is a challenging task, which currently requires the physical availability of such system in order to measure their performance. However, a model-based performance approach would allow to predict the system's performance already during early design and can thereby shorten the design cycle considerably.

Interventional X-ray (iXR) systems are MIS that dynamically record high quality images of a patient, based on X-ray beams. Design decisions in this domain are of various kinds, such as the possibility of merging of subsystems onto one platform, moving functionality from one to another platform, and assessing whether the system is robust against minor hardware changes. This paper investigates the use of a model-based approach to obtain insight in system performance.

We have decided to build iDSL, a domain specific language and toolbox for performance evaluation of Medical Imaging Systems, on top of MoDeST [8], which recently has been extended to support the modelling and analysis of Stochastic Timed Automata (STA) using PRISM [17] and UPPAAL [18] as well as discrete-event simulation using MODES. This

* This research was supported as part of the Dutch national program COMMIT, and carried out as part of the Allegio project under the responsibility of the Embedded Systems Innovation group of TNO, with Philips Medical Systems B.V. as the carrying industrial partner.

Medical Cyber Physical Systems – Medical Device Interoperability, Safety, and Security Assurance (MCPS'14).
Editors: Volker Turau, Marta Kwiatkowska, Rahul Mangharam, and Christoph Weyer; pp. 80–93

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

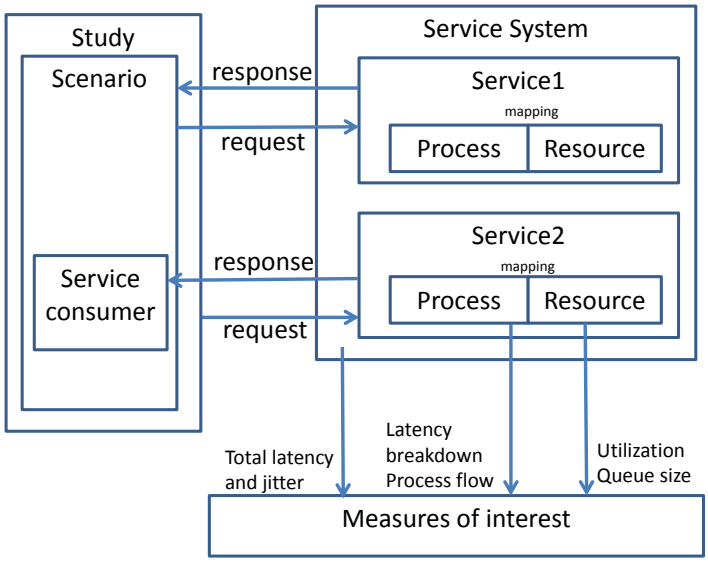

Figure 1 Conceptual model of a service system. Measures of interest are obtained using scenarios.

decision has been taken because of the expressiveness of STA and because MoDeST allows to use both analytical and simulation techniques.

We have designed the Domain Specific Language iDSL tailored towards MIS. iDSL adheres to the Y-chart philosophy [15], which separates the application from the underlying computing platform. It further uses hierarchical structures like the performance evaluation tool HIT [3]. And finally, iDSL can automatically generate design alternatives. We have constructed automated transformations from iDSL to different MoDeST model variants, each taking full advantage of the capabilities of the underlying evaluation tools, i.e., PRISM, UPPAAL and MODES. While these tools have been used widely for performance evaluation of embedded systems [13, 12, 16], to the best of our knowledge they have not been used for evaluating the performance of MIS. Finally, we use GraphViz [7] and GNUplot [20] to present performance outcomes graphically.

As for related work, [14, 19] apply model checking with UPPAAL on real-time medical systems to address safety. A study in which PRISM is used, addresses quantitative verification of Implantable Cardiac Pacemakers [5], which are time critical systems. [11, 21] evaluate the performance of iXR systems based on the Analytical Software Design (ASD) method.

The Octopus Toolset [2] provides various tools for the modelling and analysis of software systems in general, whereas iDSL is specifically designed for MIS. Earlier work [10, 22] proposed a simulation-based approach using POOSL [6], leading to average values.

This paper is further organized as follows. Section 2 describes the conceptual model of iDSL. Section 3 specifies the constructs and relations that constitute the iDSL language. Section 4 covers the functionality and usage of the iDSL tool. Section 5 concludes the paper.

2 Conceptual model

This section describes the conceptual model that forms the basis of iDSL (see also Figure 1).

A **service system**, as depicted in the upper right block, provides services to service consumers in its environment. A consumer can send a request for a specific service at a certain time, after which the system responds with some delay.

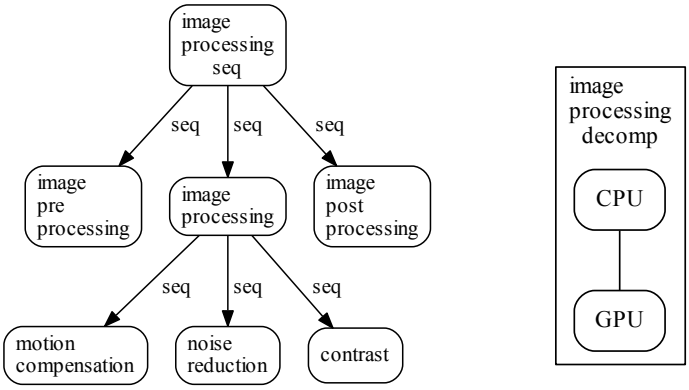

Figure 2 The IP ProcessModel (left) and IP ResourceModel (right) visualization are automatically generated from the iDSL code.

A **service** is implemented using a process, resources and a mapping, in accordance with the Y-chart philosophy [15]. A **process** decomposes high-level service requests into atomic tasks, each assigned to resources through the **mapping** (from which we abstracted in the figure). Hence, the mapping forms the connection between a process and the resources it uses. **Resources** are capable of performing one atomic task at a time, in a certain amount of time. When multiple services are invoked, their resource needs may overlap, causing concurrency and making performance analysis more challenging.

A **scenario** consist of a number of invoked service requests over time to observe the performance behaviour of the service system in specific circumstances. We assume service requests to be functionally independent of each other. That is, service requests do not affect each other's functional outcomes, but may affect each other's performance implicitly.

A **study** evaluates a selection of systematically chosen scenarios to derive the system's underlying characteristics. Finally, **measures of interest** define what performance metrics are of interest, given a system in a scenario. Measures can either be external to the system, e.g., throughput, latency and jitter, or internal, e.g., queue sizes and utilization.

3 Language constructs

We now demonstrate how to use iDSL by implementing an example Image Processing (IP) system. We have included the grammar of the iDSL language as reference at the end of the paper (see Figure 9). The iDSL language contains six sections, i.e., *Process*, *Resource*, *System*, *Scenario*, *Measure* and *Study*. The former three sections specify the functioning of the service system, whereas the latter three sections describe the way the system performance is assessed. iDSL transforms automatically into MoDeST [8] models and we therefore define its semantics in terms of MoDeST code. In what follows, we provide an iDSL instance per section, and the belonging MoDeST code that serves as semantics. In some cases, iDSL also provides an automatically generated visualization using GraphViz.

3.1 Process

A process decomposes a service into a number of atomic tasks, implemented in iDSL using a recursive data structure with layers of sub-processes. At the lowest level of abstraction, the

Table 1 Process: iDSL and MoDeST code.

iDSL Process code

```
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 }
```

Generated MoDeST Process code

```
process image_processing(){
  motion_compensation(44);
  noise_reduction(Uniform(80,140));
  contrast(134) }
process image_processing_seq(){
  image_pre_processing(50);
  image_processing();
  image_post_processing(25) }
process image_processing_application_instance(){
  generator_image_processing_application?;
  image_processing_seq() }
```

Table 2 ResourceModel: iDSL and MoDeST code.

iDSL ResourceModel code

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

Generated MoDeST ResourceModel code

```
process machine_call_GPU(real taskload){
  machine_GPU_start! {= sync_buffer=taskload =};
  machine_GPU_stop? }
process machine_GPU(){
  real taskload;
  machine_GPU_start? {= taskload=sync_buffer =};
  delay (taskload / 5 )
  machine_GPU_stop!;
  machine_GPU() }
```

atomic tasks each have a load, i.e., an amount of work, such as the number of CPU cycles.

The process for the example (Table 1 and Figure 2, left) combines *seq* and *atom* constructs. At its highest level, it consists of a sequential task that decomposes into an atomic task "pre-processing" with load 50, a sequential task "processing" and an atomic task "post-processing" with load 25. At a lower level, the sequential task "processing" consists of 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], at execution time.

In MoDeST, these hierarchies are implemented using layered processes, and the loads as parameters that are used later. The process is triggered via a generator through binary communications.

iDSL additionally supports the process algebraic constructs for parallelism (*par*), non-deterministic choice (*alt*), probabilistic choice (*palt*) and abstraction, as well as a mutual exclusion (*mutex*) to permit at most one process instance at a time on a certain process part.

3.2 Resource

In iDSL, a resource is defined as recursive hierarchical structure consisting of *decomp* and *atom* constructs, and a binary relation that defines which resources are connected.

The *decomp* construct is used to create decomposable resources, whereas the *atom* construct is used to specify atomic resources. They have a rate that specifies how much load they can process per time unit, e.g., the number of CPU cycles per second. Resources that are connected can perform operations in sequence for one process. The connections further enhance the way resources are visualized and enable high-level input validations.

We model the resource in our example as a composite resource (Table 2 and Figure 2, right). It consists of two atomic resources, i.e., a "CPU" with rate 2 and a "GPU" with rate 5. Additionally, the "CPU" and "GPU" are connected. In the MoDeST code, two processes per resource are created of which we have included the "GPU". A resource is implemented using binary communications to handle concurrency and a delay to represent the resource being in

Table 3 System: iDSL and MoDeST code.

iDSL System code

```
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) }
```

Generated MoDeST System code

```
process motion_compensation(real taskload){
  machine_call_CPU(taskload) }
process image_post_processing(real taskload){
  machine_call_GPU(taskload) }
```

Table 4 Scenario: iDSL and MoDeST code.

iDSL Scenario code

```
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,
```

Generated MoDeST Scenario code

```
process init_generator_image_processing_service
  () { delay (0)
      generator_image_processing_service() }
process generator_image_processing_service(){
  clock c; tau {= c=0 =};
  alt{
   :: generator_image_processing_application!
   :: delay(1) tau // time-out };
  when urgent(c >= (400-0) )
    generator_image_processing_service() }
```

use, i.e., processing a process. The self-recursion ensures that the resource runs forever. The delay is the quotient of the load and rate, e.g., CPU cycles divided by CPU cycles per second leads to seconds. The second process (with prefix machine_call) abstracts communications from the process layer. The MoDeST code reveals that concurrency is currently resolved using non-deterministic choices, in a non-preemptive manner.

3.3 System

A system consists of one or more services. In our example (Table 3), we construct an overall system with one service that combines the already defined process and resource (Figure 2). By defining an additional mapping, we connect them to form a service (Figure 3). In MoDEST, each mapping assignment results into a process that calls a resource.

3.4 Scenario

A scenario is defined as a bundle of services, on one system, that are individually requested over time (Table 4). The times of the requests are defined in terms of the first and second request, respectively 0 and 400 in the example here. Inter-request times are assumed to be constant, 400 in the example. To illustrate the modelling flexibility, we have added another set of service requests, including two *dspace* function calls that are constant within a design instance (to be explained later). In MoDeST, two processes handle the timing. The first (with prefix init) performs the initial delay once. The second then loops forever, with period of the inter-request time, triggering the process once per loop. When the service system fails to respond to a request immediately, a time-out occurs that drops the request.

3.5 Measure

Measures define what performance metric(s) one would like to obtain, given a system in a certain scenario. Different measures might call for different techniques to obtain them, e.g., simulation, model checking or numerical analysis. To illustrate our approach, we specified two measures (Table 5), based on two methods, i.e., MODES [8] based simulations and

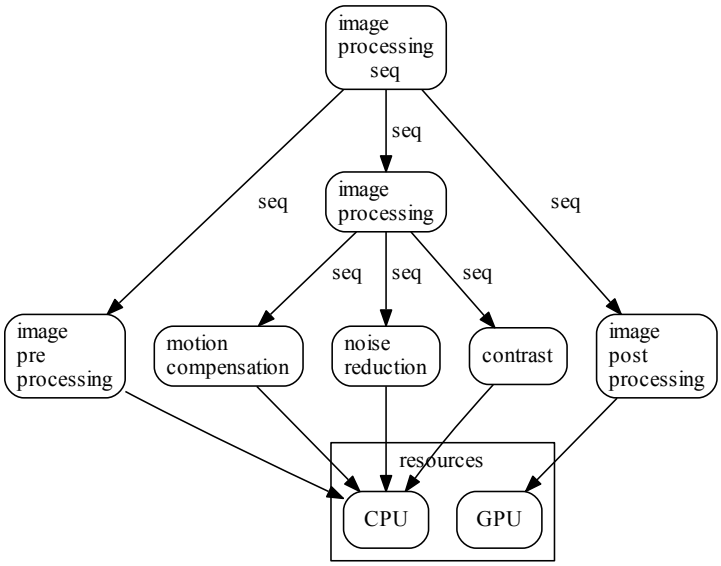

Figure 3 The IP Service visualization, which is automatically generated from the iDSL code.

UPPAAL [18] based model checking. The former uses Stochastic Timed Automata (STA) as its underlying model, while the latter uses Timed Automata (TA). We create specific MoDeST code (Table 5) for each case to combine the STA's expressiveness and the TA's model checking capability.

First, **simulations** provide response times, for a given number of simulations of a certain length. We use 1 run of length 280 in the example. Simulations additionally provide insight in resource utilizations and latency breakdowns. To eliminate non-determinism, we use an as soon as possible (ASAP) scheduler for time, and a uniform resolution for choice [9], which are fixed parameters that iDSL provides to MODES. The ASAP scheduler makes sure that whenever an action is possible, it is performed immediately. The uniform resolution selects one out of multiple actions to perform when their underlying distribution is not specified, with equal probabilities.

In MoDeST, we extend the already given code with both measurement points and properties, for both the latencies and utilizations. Each (sub)process is enclosed by a stopwatch to register a latency value, whereas an actual property retrieves this value for a single latency. Resources are augmented with a cumulative delay counter, retrieved by means of a property after some time, viz., an arbitrary 10000 in the example.

Table 5 Measure: iDSL and MoDeST code.

iDSL Measure code

```
Section Measure
  Measure ServiceResponse times
    using 1 runs of 280 ServiceRequests
  Measure ServiceResponse absolute times
    for any ServiceRequest
```

Generated MoDeST Measure code for MODES.

```
process image_processing(){
  tau {= stopwatch_image_processing = 0,
         image_processing_done = false =};
  ...
  tau {= image_processing_done = true,
         counter_image_processing++ =};
  tau {= image_processing_done = false =} }
property property_latency_image_processing =
  Xmax( stopwatch_image_processing |
        stopwatch_image_processing_done &&
        counter_image_processing==1 );
process machine_GPU(){ ...
  delay ( taskload/ 5 )
  tau {= util_counter_GPU+= ( taskload / 5) =};
  .. }
property property_utilization_CPU =
  Xmax ( util_counter_CPU/10000 | time==10000 );
```

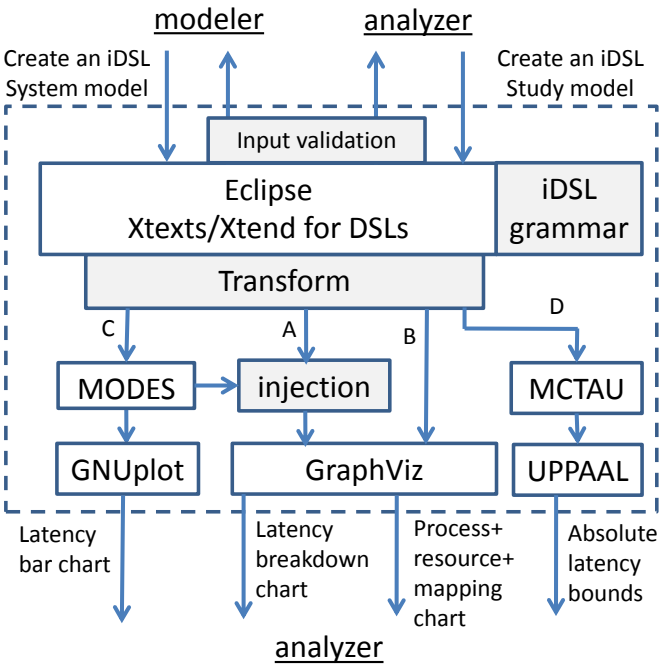

Figure 4 The iDSL tool chain overview. A modeller and analyser create an iDSL model based on the iDSL grammar. The iDSL tool transforms this model into MoDeST and GraphViz models, leading to performance measures to be evaluated by the analyser.

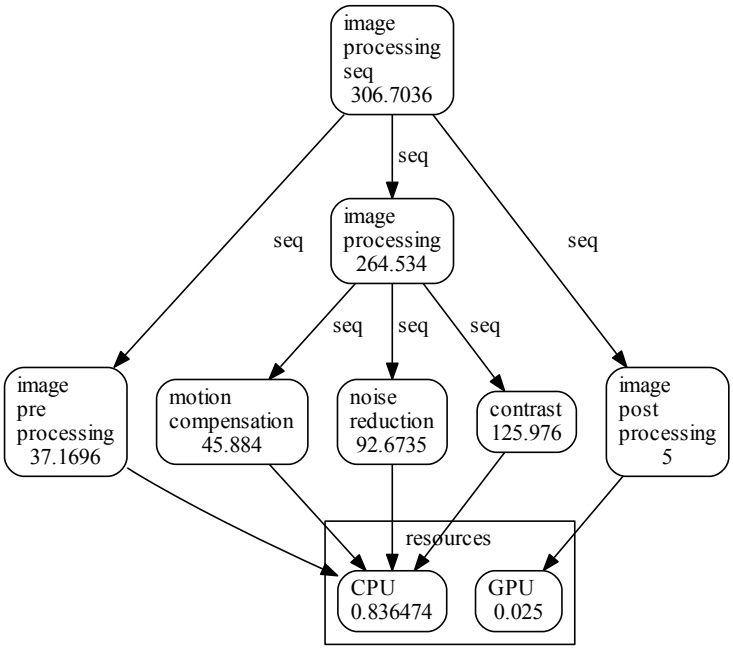

Figure 5 The latency breakdown chart and utilization (offset=0), based on MODES simulation results, which is automatically generated from the iDSL code.

Table 6 Binary search for bounds, pseudo code.

LB: Compute lower bounds, pseudo code

```
LB (lbound,ubound){
   if (ubound==lbound) return lbound
   check_value=(lbound+ubound)/2
   UPPAAL (p = probability(latency<check_value))

   if ( p=0 ) LB (check_value,ubound)
   else       LB (lbound,check_value) }
```

Compute lower bounds, execution trace

```
LB(0,1024)  -> LB(0,512)   -> LB(0,256) ->
LB(128,256) -> LB(128,192) -> LB(128,160) ->
LB(144,160) -> LB(152,160) -> LB(156,160) ->
LB(158,160) -> LB(159,160) -> LB(159,159) -> 159
```

Table 7 Study: iDSL and MoDeST code.

iDSL Study DSL code

```
Section Study
  Scenario image_processing_run
    DesignSpace
      (offset {0, 20, 40, 80, 120, 160, 200) }
```

Generated MoDeST Study code

```
real sync_buffer;
closed par{
 :: do{image_processing_application_instance()}
 :: do{image_processing_application_instance2()}
 :: init_generator_image_processing_service()
 :: init_generator_image_processing_service2()
 :: machine_CPU()
 :: machine_GPU() }
```

Second, **model checking** leads to the absolute minimum and maximum response times, given a system and scenario. It does not require parameters in the iDSL language, because its results are universal. The lower and upper bounds are *valid* when they are respectively lower and higher than all possible outcomes. They are *strict* when additionally the distance between them is minimal, i.e., the lower bound is the highest valid one and vice versa. iDSL can return bounds that are both valid and strict.

For model checking, iDSL "downgrades" STAs to TAs [9] automatically, thereby, replacing real numbers by integers, probabilistic choice and infinite distributions by non-deterministic choice, and removing some performance measuring variables to reduce the state-space size. For instance, the uniform function in the process (see Table 1), represented by a continuous probability function in STAs, becomes a non-deterministic, finite choice.

While TAs only support properties with boolean expressions, the absolute values cannot be retrieved using single properties. Therefore, we have equipped iDSL with a binary search algorithm that leads to a solution in $\mathcal{O}(\log(n))$, with n the size of the search range. The algorithm consists of two functions, i.e., a LB function to compute lower bound values and a UB function for higher bound values.

LB is a recursive function (Table 6, top) with two parameters, the lower and upper bound of the current range of values. The stop criterion, i.e., the lower and upper bound value are the same, ends the recursion by returning the lower bound value. Otherwise, the range is halved in two parts by taking the average value of the lower and upper bound. UPPAAL is queried with this value to determine in which half of the range the lower bound is located. A recursive call of LB then takes place using the right range half as parameter. The UB function operates in a similar fashion.

To illustrate the functioning of LB, we apply it on the case with one image processing system. We start by selecting the initial range of values. Since the algorithm is of $\mathcal{O}(\log(n))$ and the choice of n does therefore not affect the workload much, it is advised to overestimate the size of the range. Based on simulation results, we choose [0:1024] to be our initial range. The execution trace (Table 6, bottom) conveys 12 recursive calls before the final value 159 is finally obtained. This means that the one image processing system will never display a service response time smaller than 159, in the given scenario.

3.6 Study

Finally, a study forms a collection of scenarios that one would like to analyse in an automated manner. This is principally done by summing up one or more scenarios (Table 7). Individual MoDeST models are created for each scenario, which each contain a main parallel process to initiate all process model threads, generators and the resources involved.

We conclude with a design space, a shorthand way to specify a set of similar scenarios. In our example, we vary the starting time offset of one of the service-request sequences to be 0, 20, 40, 80, 120, 160 and 200. For this purpose, we create a design space in the study and enumerate the desired values. After this, the *dspace* function can be used for the offset parameter as done in the system section (Table 3). As a result, seven similar scenarios, one corresponding to each offset value, are created that vary where and only where the *dspace* function is used.

4 Tool and solution chain

This section covers the functioning of our iDSL tool (illustrated in Figure 4). iDSL requires two user roles to be fulfilled, the *modeller* and the *analyser*. The modeller constructs a model of a real system and the analyser specifies measures to perform. Execution of the model then generates artefacts with performance metrics to be investigated by the analyser.

4.1 Modelling

A modeller and analyser interactively create an iDSL instance in the Eclipse IDE for DSL Developers[1], adhering to iDSL's grammar. Input validation comprises syntax checking and advanced checks, e.g., for unique naming and non-circular definitions. Additionally, warnings and information boxes are displayed, e.g., when the design space is large. The modelling ends with the creation of a valid iDSL model.

4.2 Execution

Created iDSL models are then automatically transformed into two kinds of GraphViz specifications (Figure 4, A+B) and two kinds of MoDeST models (Figure 4, C+D). Transformations are written in Xtend and generate text output, based on iDSL instance constructs.

Some GraphViz specifications are performance unrelated and provide a visual presentation of the processes, resources and mappings of the system (as already shown in Figures 2 and 3). They are turned into a *process+resource+mapping chart* using the GraphViz tool.

The remaining GraphViz specifications have placeholders to contain performance numbers and form one input of the *injection* step. Some MoDeST models are executed in the MODES simulator and lead to latency and utilization numbers. The high-level latencies per instance are transformed into a *latency bar chart* by GNUplot.

The latencies at different process levels and utilizations form the second input of the *injection*. The remaining MoDeST models are executed in UPPAAL, via MCTAU [4], to obtain *absolute latency bounds*.

The injection step takes a GraphViz specification with placeholders and MODES performance numbers as input. By simply injecting the performance numbers at the right placeholders, a new GraphViz specification results. It is forwarded to the GraphViz tool and transformed into a *latency breakdown chart* (Figure 5). To illustrate the meaning of this

[1] `http://www.eclipse.org/downloads/packages/eclipse-ide-java-and-dsl-developers/junosr2`

chart, we show that the latency of a sequential process equals the sum of its sub-processes' latencies, e.g. for "image processing", the latency is (rounded off): $265 = 46 + 93 + 126$. Additionally, the utilization is the quotient of the busy time of a resource and the total elapsed time. Take for instance the "GPU" resource, which is only used for "image processing" and for 5 time units per service. Services are each invoked periodically every 400 time units. Therefore, the "GPU" has a utilization of $(5 + 5)/400 = 0.025$.

4.3 Analysis

Using the presented tool chain, iDSL offers the possibility to compare several design alternatives from various perspectives, in an automated manner. We proceed with discussing the results iDSL can generate. First, we discuss the results based on MODES simulations. After that, we review results obtained from model checking using MCTAU and UPPAAL.

Simulation results. We have defined a study with seven design alternatives for which iDSL automatically generates a latency breakdown chart and a latency bar graph. We present the ones for the offset=0 case (Figures 5 and 6). As can be seen in Figure 6, the latency varies highly. This is due to a high degree of concurrency, which forces the scheduler to make many concurrency resolving decisions that each increase the variability. We further see that the "noise reduction" and "contrast" processes contribute most to the latency, which stems directly from their large loads. Additionally, we have included a CDF with the latency times of the design alternatives altogether (Figure 7). It shows that when the offset is small and the level of concurrency larger, latency times become higher. For the highest offsets, no concurrency takes place.

Model checking results. We applied MCTAU on the case with one image processing system. The computation of the lower (Table 6, bottom) and upper bound leads to values 159 and 189, respectively. The difference of 30 between them is caused by the uniform distribution that is specified in the "noise reduction" process (Figure 8). As required by definition, all simulation outcomes fall within the absolute bounds.

5 Conclusion and future work

In this paper we presented iDSL, a domain specific language and toolbox for the performance evaluation of Medical Imaging Systems. iDSL automates performance analysis, for both model checking and simulations, and displays results visually. We have demonstrated the feasibility of our approach using a small example based on a real system, in which we investigated MIS with two concurrent image processing applications.

iDSL has successfully returned differentiated delay, utilization and bound values for a number of designs. In order to assess the scalability of iDSL, we will apply it on extensive cases of our industrial partner Philips, in the Allegio project[2]. This will put the expressiveness of the iDSL language to the proof and may lead to extensions to both the language and toolbox.

We are currently investigating whether we can add a transformation for probabilistic model checking. To support analysis further, we will extend iDSL to create graphs and diagrams that display information of multiple scenarios, services and simulation runs, and include GANTT charts.

[2] `http://redesign.esi.nl/research/applied-research/current-projects/allegio/`

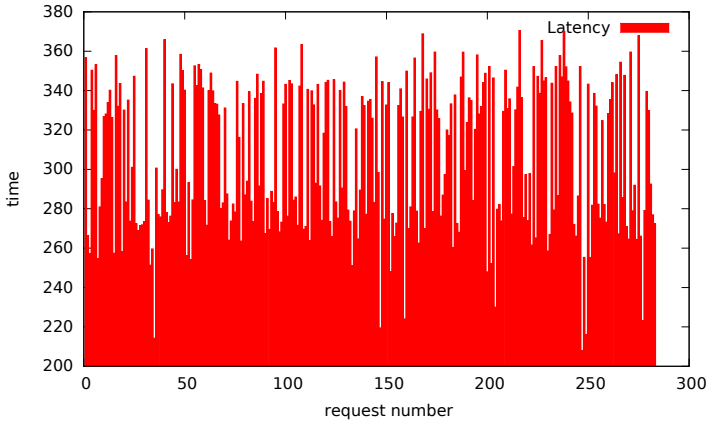

Figure 6 The MODES latency times bar graph (offset=0) for 280 service requests, which is automatically generated from the iDSL code.

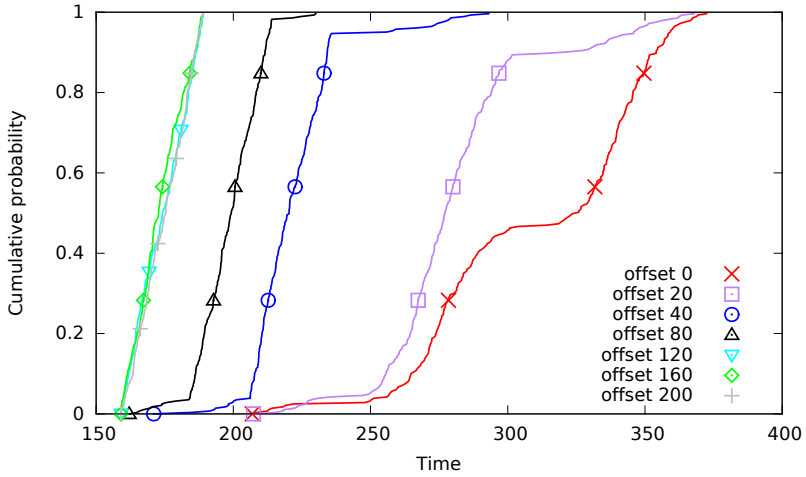

Figure 7 The cumulative distribution functions of latencies for seven design instances is automatically generated from the iDSL code.

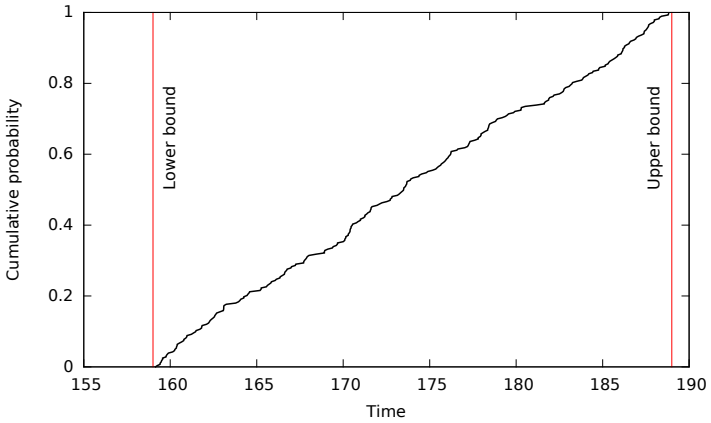

Figure 8 The absolute minimum and maximum bounds, and a CDF of the simulation outcomes is automatically generated from the iDSL code.

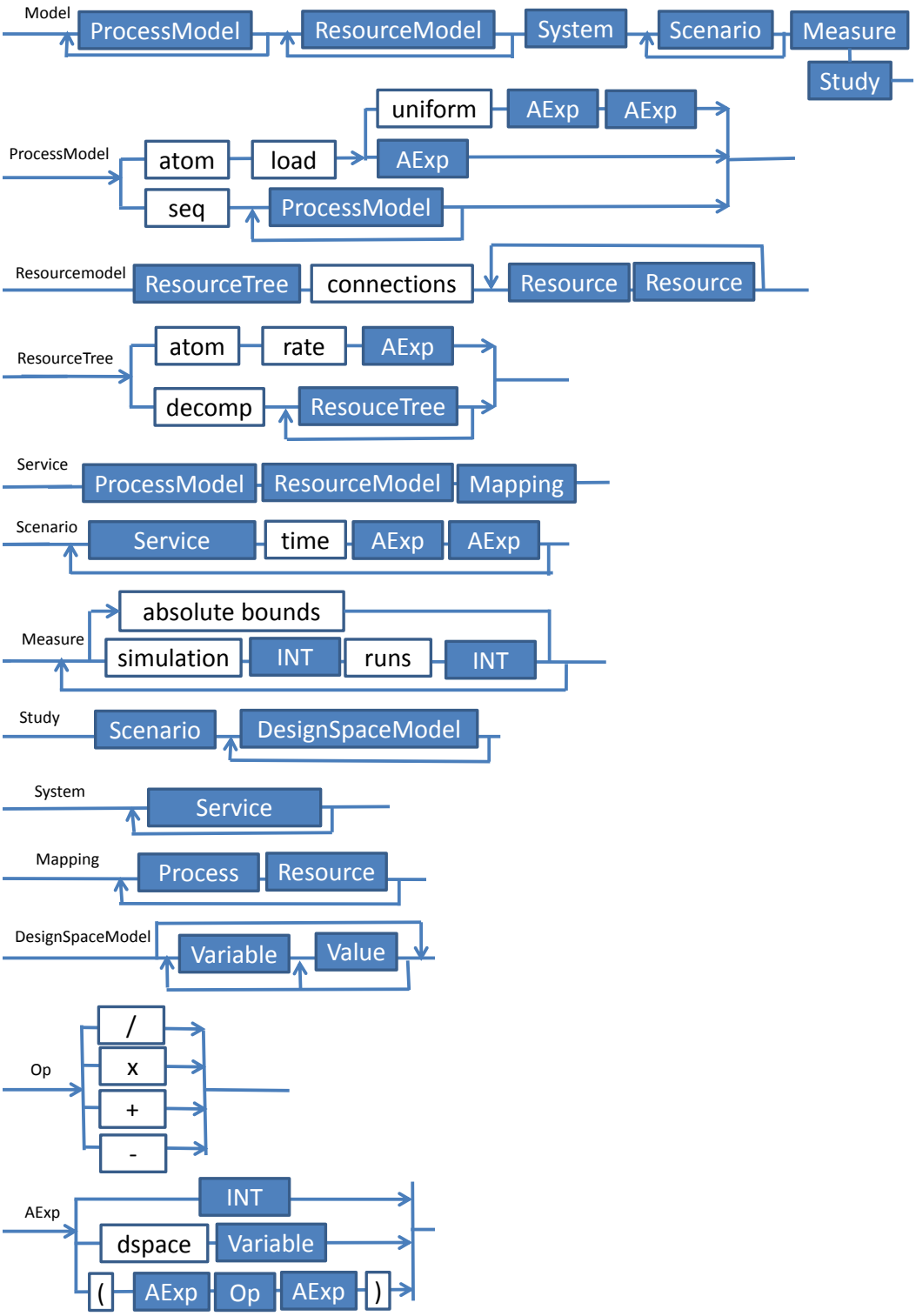

Figure 9 The grammar of iDSL's language as used in this paper. The grammar has the *Model* concept as its top-level node. It decomposes into of one or more *ProcessModels*, *ResourceModels*, a *System* (a set of *Services*), *Scenarios*, and a *Measure* and a *Study*.

Acknowledgements. We would like to thank Arnd Hartmanns of the MoDeST development team for his help and efforts made during the development of iDSL.

We would like to thank Arjan Mooij of the Allegio project for informing us about the binary search algorithm.

References

1 H. Alemzadeh, R. Iyer, Z. Kalbarczyk, and J. Raman. Analysis of safety-critical computer failures in medical devices. *IEEE Security & Privacy*, 11(4):14–26, 2013.

2 T. Basten, E. Van Benthum, M. Geilen, M. Hendriks, F. Houben, G. Igna, F. Reckers, S. De Smet, L. Somers, and E. Teeselink. Model-driven design-space exploration for embedded systems: the Octopus toolset. In *Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6415 of *LCNS*, pages 90–105. Springer, 2010.

3 H. Beilner, J. Mater, and N. Weissenberg. Towards a performance modelling environment: News on HIT. In *Modeling Techniques and Tools for Computer Performance Evaluation*, pages 57–75. Plenum Press, 1989.

4 J. Bogdoll, A. David, A. Hartmanns, and H. Hermanns. MCTAU: Bridging the gap between modest and UPPAAL. In *Proc. 19th International SPIN Workshop on Model Checking of Software*, volume 7385 of *LNCS*, pages 227–233. Springer, 2012.

5 Taolue Chen, Marco Diciolla, Marta Kwiatkowska, and Alexandru Mereacre. Quantitative verification of implantable cardiac pacemakers. In *Proc. 33rd Real-Time Systems Symposium*, pages 263–272. IEEE, 2012.

6 Eindhoven University of Technology. Software/Hardware Engineering - Parallel Object-Oriented Specification Language (POOSL). http://www.es.ele.tue.nl/poosl/.

7 J. Ellson, E. Gansner, L. Koutsofios, S. North, and G. Woodhull. Graphviz—open source graph drawing tools. In *Graph Drawing*, volume 2265 of *LNCS*, pages 483–484. Springer, 2002.

8 E. Hahn, A. Hartmanns, H. Hermanns, and J.-P. Katoen. A compositional modelling and analysis framework for stochastic hybrid systems. *Formal Methods in System Design*, 43(2):191–232, 2012.

9 A. Hartmanns. Model-checking and simulation for stochastic timed systems. In *Proc. 9th International Symposium on Formal Methods for Components and Objects*, volume 6957 of *LCNS*, pages 372–391. Springer, 2010.

10 S. Haveman, G. Bonnema, and F. van den Berg. Early insight in systems design through modeling and simulation. In *Proc. 12th Annual Conference on Systems Engineering Research*, 2014. To appear.

11 S. Hettinga. Performance analysis for embedded software design. *Master's thesis, University of Twente*, 2010.

12 G. Igna, V. Kannan, Y. Yang, T. Basten, M. Geilen, F. Vaandrager, M. Voorhoeve, S. de Smet, and L. Somers. Formal modeling and scheduling of datapaths of digital document printers. In *Formal Modeling and Analysis of Timed Systems*, volume 5215 of *LCNS*, pages 170–187. Springer, 2008.

13 G. Igna and F. Vaandrager. Verification of printer datapaths using timed automata. In *Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6416 of *LCNS*, pages 412–423. Springer, 2010.

14 Z. Jiang, M. Pajic, and R. Mangharam. Cyber-physical modeling of implantable cardiac medical devices. *Proc. of the IEEE*, 100(1):122–137, 2012.

15 B. Kienhuis, E. Deprettere, P. van der Wolf, and K. Vissers. A methodology to design programmable embedded systems. In *Embedded processor design challenges*, volume 2268 of *LCNS*, pages 18–37. Springer, 2002.

16 M. Kwiatkowska, G. Norman, and D. Parker. Controller dependability analysis by probabilistic model checking. *Control Engineering Practice*, 15(11):1427–1434, 2007.

17 M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: verification of probabilistic real-time systems. In *Computer Aided Verification*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

18 K. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1):134–152, 1997.

19 M. Pajic, Z. Jiang, I. Lee, O. Sokolsky, and R. Mangharam. From verification to implementation: A model translation tool and a pacemaker case study. In *Proc. 18th Real-Time and Embedded Technology and Applications Symposium*, pages 173–184. IEEE, 2012.

20 J. Racine. GNUplot 4.0: a portable interactive plotting utility. *Journal of Applied Econometrics*, 21(1):133–141, 2006.

21 R. Sadre, A. Remke, S. Hettinga, and B.R. Haverkort. Simulative and analytical evaluation for asd-based embedded software. In *Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance*, volume 7201 of *LCNS*, pages 166–181. Springer, 2012.

22 F. van den Berg, A. Remke, A. Mooij, and B.R. Haverkort. Performance evaluation for collision prevention based on a domain specific language. In *Computer Performance Engineering*, volume 8168 of *LCNS*, pages 276–287. Springer, 2013.