A Domain-Specific Language and Toolchain for Performance Evaluation based on Measurements

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

[1] DACS, University of Twente, Enschede, The Netherlands
{ f.g.b.vandenberg | b.r.h.m.haverkort }@utwente.nl
[2] ICIS, Radboud University Nijmegen, The Netherlands
[3] Embedded Systems Innovation (ESI) by TNO, Eindhoven, The Netherlands

Abstract. This tool paper presents iDSL, a language and a fully automated toolchain for evaluating the performance of service-oriented systems. In this work, we emphasize the use of a *high-level domain specific language* that is tailored to be understood by system designers and domain experts, a transformation into an underlying process algebra which contains latency distribution functions based on *real measurements* for calibration, and the *integration of analysis tools* under the hood. Altogether, the approach delivers intuitive, visual results.

1 Motivation

Embedded systems are computer systems that have a dedicated function within a larger system, often with real-time constraints [19]. Hence, their performance is vital. However, good performance is hard to achieve, because embedded systems come with increasingly heterogeneous, parallel and distributed architectures and may comprise many product lines and different configurations.

Here, we consider service-oriented systems [10–15], a subclass of embedded systems, which: (i) provide services to their environment, accessible via so-called requests; (ii) each service request leads to one response; (iii) service requests are functionally isolated from each other; but, (iv) may affect each other's performance by competing for the same resource in the service-oriented system.

We propose a performance evaluation framework that can be used to *evaluate* the performance of service-oriented systems based on real *measurements* for calibration (Contribution $\mathcal{C}1$). We realize this framework via iDSL, which comprises the domain-specific, *high-level* iDSL language (Contribution $\mathcal{C}2$) to model service-oriented systems and the iDSL toolchain (Contribution $\mathcal{C}3$) to evaluate the performance of these systems in a fully automatic fashion. This approach separates the description of the user concerns from the solution approach, in accordance with the Declarative Performance Engineering (DPE, [16]) approach.

2 The high-level iDSL language

The iDSL language [10–15] has been developed to model service-oriented systems. It is tailored to be used and understood by system designers and experts

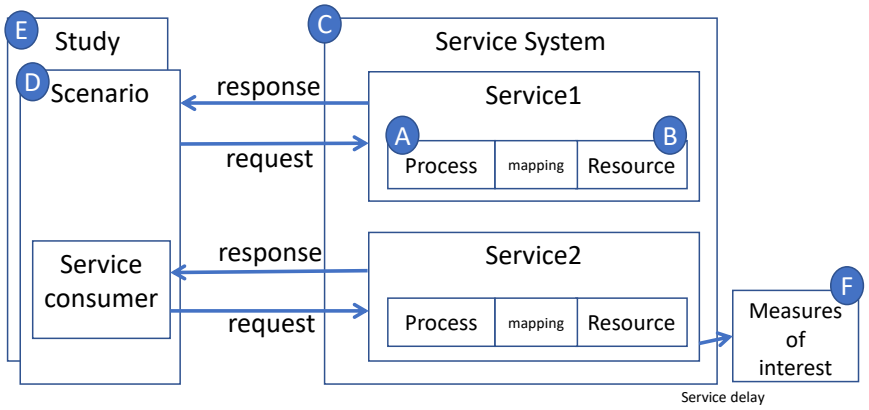


Fig. 1: The meta model of the iDSL language

in the service-oriented systems domain, in line with $\mathcal{C}2$. Figure 1 depicts the six high-level concepts of the iDSL language, as follows. A service system (Figure 1-C) provides services to 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. A service is implemented using a process (A), resources (B) and a mapping. A process decomposes high-level service requests into atomic tasks, which are each assigned to a resource in the mapping. 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 contention and making performance analysis harder. A scenario (D) consists of a number of invoked service requests over time to observe specific performance behavior of the system. A study (E) evaluates a selection of systematically chosen scenarios to derive the system's underlying characteristics. Finally, measures of interest (F) define what performance metrics to obtain, given a system in a scenario.

For illustration, Table 1 provides an example iDSL language instance of a medical imaging system [14, Section 3], as follows. The process contains a sequence of the processes "image_pre_processing", "image_processing" and "image_post_processing". In turn, process "image_processing" decomposes into "motion_compensation", "noise_reduction" and "contrast". Each atomic process has a load, an amount of work. The resource contains a CPU with rate 2, i.e., it can process 2 loads per time unit, and a GPU with rate 5. The system combines the process and resource, and has a mapping to connect atomic tasks to resources. The scenario encompasses two streams of requests for the only service. Both streams have fixed inter-arrival times of 400. One stream has an initial delay of 0. The initial delay of the other is determined by an offset parameter, which is a variable that is defined in the so-called design space of the study. Finally, the measure contains two measures of interest referring to performance evaluation.

Table 1: An example service-oriented system, modeled using the iDSL language

(a) 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 }
```

(b) Resource

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

(c) System

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

(d) 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), ...
```

(e) Study

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

(f) Measure

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

(g) Process with an injected EDF

```
Section Process
   ProcessModel normal_U100_010_n100  palt { 2 atom load 91
      1 atom load 92 1 atom load 93 2 atom load 95 5 atom load 96
      9 atom load 97 9 atom load 98 15 atom load 99 15 atom load 100
      15 atom load 101 9 atom load 102 7 atom load 103 5 atom load 104
      3 atom load 105 2 atom load 107 }
```

3 The integrated iDSL toolchain

In this section, we discuss the iDSL toolchain which ranges from creating an iDSL language instance to generating performance artifacts, in line with $\mathcal{C}3$.

Creating the performance model involves the conjoint modeling by a modeler and analyzer of a *case study* in the iDSL language. A modeler determines how the system behaves and generates a system model, i.e., a process, resource and system (cf. Figure 1-A,B,C). The analyzer determines system usage and creates a study, i.e., scenario, study and measure (cf. Figure 1-D,E,F).

During the modeling process, the Eclipse Integrated Development Environment [2] is used to support the user. This environment enables, among others, syntax highlighting, code completion, and "input validation", e.g., checking the code for invalid references, unused objects and ambiguous definitions. Also warnings and information boxes are displayed, e.g, when the design space is too large.

Under the hood, the iDSL grammar has been defined using the Xtext framework [18]. The toolchain functionality is programmed in the Xtend language [17].

In the following, we briefly describe the four main activities that constitute the performance analysis toolchain of iDSL.

Process measurements. Measurements are performed on a real system and injected the into the iDSL model for calibration [15, Section 3]. The text-processing tool AWK [1] is used to facilitate this.

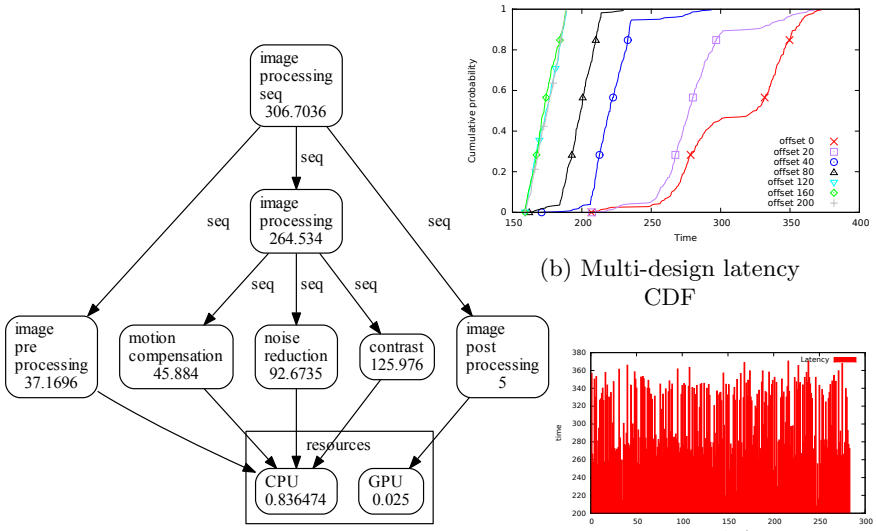
1. Perform measurements on a real system [15, Section 3.1].
2. Create Gantts: group measurements into execution times [15, Section 3.2].
3. Generate Empirical Distribution Functions (EDFs) [15, Section 3.3].
4. Inject the EDFs of step 3 into the IDSL model via a model transformation: represent EDFs as probabilistic alternatives (PALT, [4]) constructs, in line with $\mathcal{C}1$. For illustration, we have drawn 100 numbers from a normal distribution ($\mu = 100$, $\sigma = 10$) [7] representing measurements. Table 1g then shows the resulting EDF in iDSL. For instance, "2 atom load 91" means that the 100 drawn numbers contain 2 times value 91.

Model simplification. iDSL determines whether the model can practically be evaluated [12, Section 4.3]. If not, it is simplified via a transformation, as follows.

1. Cluster similar measurements in each generated EDF [12, Section 4.1].
2. Increase the time unit of all time occurrences in the model [12, Section 4.2].

Model evaluation is delegated to Modest [4].

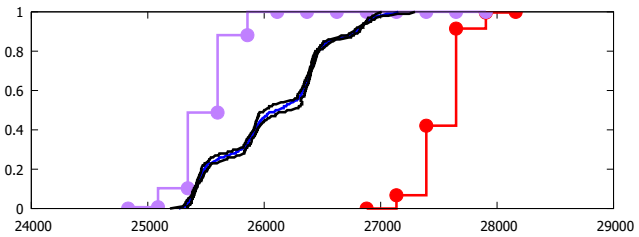
1. Create Modest models: transform iDSL into Modest [11, Section 4.3]
2. Evaluate the Modest models for performance using the Modest toolset.
   (a) Discrete-event simulation: yields average latencies [14, Section 4.2]:
   (b) Timed Automata (TA)-model checking: a binary search for absolute bounds [14, Section 4.2].
   (c) Probabilistic Timed Automata (PTA)-model checking: an iterative algorithm in which cumulative latency probabilities are computed one at a time [13, Section 4].
   (d) Efficient PTA-model checking: a carefully constructed combination of the aforementioned techniques [12, Section 6].
3. Parse results: parse the Modest results into high-level iDSL results.



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

(b) Multi-design latency CDF

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



(d) The lower (in purple) and upper bound CDF (in red), the simulation avg. (in blue), and $\alpha = 0.95$ CI (in black)

Fig. 2: Four ways of representing latencies, generated from the iDSL code

Create visualizations turns the parsed results into intuitive graphs.

1. Latency breakdown chart (see Figure 2a): displays the structure of a service, i.e., the underlying processes and resources, and its dynamics, i.e., process latencies and resource utilizations.
2. Multi-design latency Cumulative Distribution Function (CDF, see Figure 2b): provides latency CDFs for multiple designs in one graph to easily determine the effect of design decisions.
3. Latency bar chart (see Figure 2c): shows the subsequent latency times of a service which provides insight in jitter, i.e., the variation of latencies.
4. Latency CDF (see Figure 2d): provides a lower (purple) and upper bound (red) CDFs whose difference is the result of how nondeterminism is resolved.

Figure 2a till 2c are based on discrete-event simulations, and Figure 2d on PTA-model checking. Figure 2a is made by GraphViz [3], the others by GNUplot [6].

4 Background

iDSL is different from tools such as the Modest toolset [4], Storm [8], UPPAAL [9] and PRISM [5]. Where the latter deliver relatively generic, widely-applicable languages, instead, iDSL provides a *domain-specific* language ($\mathcal{C}2$) which allows measurements-based calibration ($\mathcal{C}1$), and a fully automated toolchain ($\mathcal{C}3$).

References

1. R. Andrews, D. Jones, J. Williams, P. Thorson, G. Oliver, D. Costa, and B. Le Boeuf. Heart rates of northern elephant seals diving at sea and resting on the beach. *Journal of Experimental Biology*, 200(15):2083–2095, 1997.
2. Eclipse desktop web IDEs. https://www.eclipse.org/ide/.
3. Graphviz - Graph Visualization Software. http://www.graphviz.org/.
4. A. Hartmanns and H. Hermanns. The Modest Toolset: An Integrated Environment for Quantitative Modelling and Verification. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 593–598. Springer, 2014.
5. M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: verification of probabilistic real-time systems. In *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer, 2011.
6. J. Racine. GNUplot 4.0: a portable interactive plotting utility. *Journal of Applied Econometrics*, 21(1):133–141, 2006.
7. RANDOM.ORG. https://www.random.org/gaussian-distributions/.
8. Storm Checker. http://www.stormchecker.org.
9. Uppsala Aalborg model checker. http://www.uppaal.org/.
10. F. van den Berg. *Automated Performance Evaluation of Service-Oriented Systems*. PhD thesis, University of Twente, 2017.
11. F. van den Berg, B. R. Haverkort, and J. Hooman. iDSL: Automated Performance Evaluation of Service-Oriented Systems. In *ModelEd, TestEd, TrustEd*, volume 10500 of *Lecture Notes in Computer Science*, pages 214–236.
12. F. van den Berg, B.R. Haverkort, and J. Hooman. Efficiently Computing Latency Distributions by Combined Performance Evaluation Techniques. In *Proceedings of the 9^{th} EAI International Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS'15, pages 158–163. ICST, 2015.
13. F. van den Berg, J. Hooman, A. Hartmanns, B.R. Haverkort, and A. Remke. Computing Response Time Distributions Using Iterative Probabilistic Model Checking. In *Computer Performance Engineering*, volume 9272 of *Lecture Notes in Computer Science*, pages 208–224. Springer, 2015.
14. F. van den Berg, A. Remke, and B.R. Haverkort. A Domain Specific Language for Performance Evaluation of Medical Imaging Systems. In *5th Workshop on Medical Cyber-Physical Systems*, volume 36 of *OpenAccess Series in Informatics*, pages 80–93. Schloss Dagstuhl, 2014.
15. F. van den Berg, A. Remke, and B.R. Haverkort. iDSL: Automated Performance Prediction and Analysis of Medical Imaging Systems. In *Computer Performance Engineering*, volume 9272, pages 227–242. Springer, 2015.
16. J. Walter, A. van Hoorn, H. Koziolek, D. Okanovic, and S. Kounev. Asking what?, automating the how?: The vision of declarative performance engineering. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, pages 91–94. ACM, 2016.
17. Xtend. https://www.eclipse.org/xtend/.
18. Xtext. https://www.eclipse.org/Xtext/.
19. R. Zurawski. *Embedded Systems Handbook*. CRC Press, 2005.