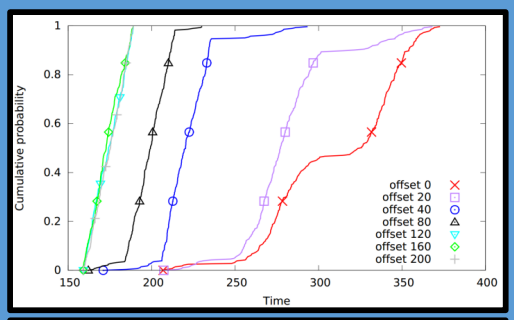
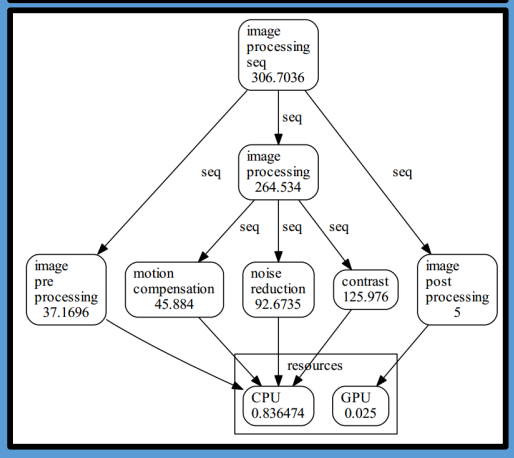
Automated Performance Evaluation of Service-Oriented Systems

```
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 }

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

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) }


Section Scenario
 Scenario image_processing_run
  ServiceRequest image_processing_service
   at time 0, 400, ...


Section Measure
 Measure ServiceResponse times
  using 1 runs of 280 ServiceRequests
 Measure ServiceResponse absolute times


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





Freek van den Berg

AUTOMATED PERFORMANCE EVALUATION OF SERVICE-ORIENTED SYSTEMS

DISSERTATION

to obtain
the degree of Doctor at the University of Twente,
on the authority of the Rector Magnificus,
Prof. dr. T.T.M. Palstra
on account of the decision of the Graduation Committee,
to be publicly defended
on Wednesday, June 14th, 2017 at 14:45

by

Fredericus Gerrit Brand van den Berg
(Freek van den Berg)

born on October 23rd, 1980
in Nijmegen, The Netherlands

Graduation committee:

Chairman:
Prof. dr. P.M.G. Apers — University of Twente

Supervisors:
Prof. dr. ir. B.R.H.M. Haverkort — University of Twente
Prof. dr. J.J.M. Hooman — TNO-ESI, Radboud University Nijmegen

Members:
Prof. dr. A.K.I. Remke — University of Twente
Prof. dr. J.L. van den Berg — University of Twente
Prof. dr.-ing. H. Hermanns — Saarland University
Prof. dr. ir. J.P.M. Voeten — Eindhoven University of Technology
Prof. dr.-ing. M. Siegle — Bundeswehr University Munich

CTIT

CTIT Ph.D.-thesis Series No. 17-434
Centre for Telematics and Information Technology
University of Twente
P.O. Box 217, 7500 AE
Enschede, The Netherlands

ISBN 978-90-365-4359-0
ISSN 1381-3617
DOI 10.3990/1.9789036543590

COMMIT/ ALLEGIO

Embedded Systems Innovation BY TNO

PHILIPS Healthcare

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

AUTOMATED PERFORMANCE EVALUATION OF SERVICE-ORIENTED SYSTEMS

Preface

After receiving my master degree in Information Science at the Radboud University of Nijmegen, I applied for a couple of software engineering jobs. However, the people doing the job interviews advised me, on the basis of my way of thinking and acting, to look for a job in a research setting. That is what I did. After that, A PhD position in performance evaluation at the University of Twente caught my eye for which I got accepted. In the five years that followed, I regularly visited both the university to improve my scientific skills as well as the premises of industrial partner Philips Healthcare to conduct case studies on real medical machines. All these efforts resulted in a software product named iDSL that automates the performance evaluation process, a number of scientific publications, and most importantly the thesis you are holding right now.

In the remainder of this preface, I will show my gratitude towards the people who made it possible to make these accomplishments. In case I did not mention you, it is because there are so many people to thank. Either way, I very much appreciate every bit of support you gave me.

Let me start by saying that I feel lucky to have had two skilled promotors by my side with completing qualities, who have enabled me to fulfill the PhD trajectory as good as possible. On the one hand, I would like to thank Boudewijn Haverkort for focusing on long-term solutions. He has allowed me to work on the large software project named iDSL without demanding immediate results. On the other hand, Jozef Hooman selflessly took care of me when my work was in need of some guidance. We met frequently to make sure that my short term results were concise and focused.

Regarding the Design and Analysis of Communication Systems department (DACS) at the university, I would like to express my gratitude to secretary Jeanette de Boer for repeatedly helping me out with the oh-so important but underestimated administrative procedures; your door is literally always open. Giovane Mauro, thank you for the template of your thesis; you have saved me a lot of work. Thank you Bjorn Postema, it has been a good learning experience to publish a paper with you; we managed to cooperate well despite our different

styles of working and thinking. Anne Remke, thank you for guiding me initially and thereby preparing me for what science has to offer. Finally, I would like to thank the other members of DACS for the pleasant working atmosphere, the interesting coffee talks, and for showing up at the defense of this thesis.

With respect to Philips Healthcare, I thank Mathijs Visser for being of prime importance of making this thesis a success. He has provided me measurement values of and insight in real medical machines, which allowed me to model these machines in great detail. Mathijs has always made time for me, even though he was very busy working for Philips as well. I would also like to Alex Admiraal for answering additional questions about the medical machines. Furthermore, Hans Driessen, Jan Stevens, Pascal Wolkotte, Robert Huis in 't Veld and Rob Albers are thanked for helping me in various ways.

Also at the premises of Philips, I constantly shared a room with three other PhD candidates who also conducted case studies on medical machines, but from different perspectives. Sarmen Keshishzadeh, Nicholas Dintzner and Steven Haveman, it has been fun sharing an office with you. We managed to have interesting discussions even though our fields of expertise did not always completely overlap. Steven, thank you for appreciating the under-the-hood work I did for your paper and making me co-author of it.

As to Embedded Systems Innovation by TNO, I would like to show my gratitude to Dave Watts for sharing his rich working experience with me, often accompanied by useful anecdotes. Arjan Mooij, thank you for helping me out with my presentations and papers in a playful yet effective way.

My deepest gratitude also goes to Arnd Hartmanns of the Modest Toolset team for providing me his full support while I made use of his toolset. You have constantly been available to me to answer my Modest and process algebra-related questions as well as quickly fixed bugs in Modest that my code conveyed. Consequently, you have become a co-author of one of my papers.

Moreover, I would like to thank the committee members for their participation in the committee and their suggestions for improvement. Particularly, Jeroen Voeten and Markus Siegle, you have sent me a lot of feedback.

Finally, I would like to thank my family for providing me moral support in general and specifically by attending the defense of this thesis. In particular, Annie Polvora, with whom I am almost six year in a relationship right now, and my parents, Gerrit and Maria, who have supported me in my private life and thereby enabled me to fully focus on my professional life.

Enschede, 17-5-2017
Freek van den Berg

Summary

An embedded system is a software-intensive system that has a dedicated function within a larger system. Embedded systems range from small portable devices, such as digital watches and MP3 players, to large complex systems, like road vehicles and medical equipment. Embedded systems have increased significantly in complexity over time and are confronted with stringent cost constraints. They are frequently used to perform safety critical tasks and their malfunctioning can lead to serious injury or even fatalities, as with medical systems.

Embedded systems interact with their environments in a time-critical way. Hence, their *safety* is predominantly determined by their performance. Performance is the amount of work a system accomplishes over time, which is frequently expressed in, among others, terms of response times, throughputs, utilizations and queue sizes. Good performance is hard to achieve, because: (i) performance evaluation is hardly ever an integral part of software engineering; (ii) system architectures are increasingly heterogeneous, parallel and distributed; (iii) systems are often designed for many product families and different configurations; and, (iv) measurements that gain insight in system performance tend to be expensive to obtain.

In this thesis, we consider so-called *service(-oriented) systems*, a special class of embedded systems. A service system provides services to its environment, which are accessible via so-called service requests, and generates exactly one service response to each request. In a service system, service requests are functionally isolated, but can affect each other's performance negatively due to competition for shared resources. Evaluating the performance of service systems before they are fully implemented is hard; answering performance questions in this domain calls for models in which timing aspects and mechanisms to deal with uncertainty are combined. Examples of uncertainty are the use of statistical predictions for execution times, two parallel processes competing to use a resource first, and system parts that are only partially specified in the model.

We propose a new performance evaluation framework for service systems. The system designer models the performance of a system, leading to a *high-level performance model*. Under the hood, this model is transformed into an *underlying performance model*, which is *evaluated* for performance. This leads to *performance results*, which are, in turn, presented to the system designer.

The literature reports about many so-called toolsets that facilitate the performance evaluation process for the system designer. A toolset is a collection of connected tools that automates (part of) the performance evaluation process. The majority of the toolsets, however, require the user to have knowledge of formal performance evaluation techniques. A system designer normally does not have this knowledge, because these techniques are not part of his daily practice. On top of that, many toolsets provide a high-level model that is *not* domain specific. This leads to a large modeling effort, and hinders the understanding and communicability of the model. Morever, most toolsets tend to return approximate results, because the generation of exact results generally relies on exhaustive evaluations. Finally, it is not uncommon that the system designer needs to perform labor-intensive and error-prone efforts to aggregate and visualize results.

The goal of this thesis is to overcome the above shortcomings; the system designer should be able to create a performance model in a language of his domain and obtain results in terms of this domain. This calls for a *high-level model* that is domain specific, expressive and concise. The model should also explicitly separate the application (software) from the platform (hardware) to support the independent evaluation of combinations of them. Also, the model should facilitate calibrations based on real measurements to make the model more realistic. Moreover, *performance results* should be presented in a visual, comprehensive way and cover a variety of performance metrics, e.g., utilizations, queue sizes, throughputs and latencies, to gain a deep insight in the underlying system's performance characteristics. Finally, the system designer should be able to perform *fully automated evaluations*, for many designs and modes of analysis, and within a practical amount of time.

To achieve the goal of this thesis, we have designed and implemented iDSL, a language and toolchain for performance evaluation of service systems. Interaction with system designers conveyed that they find the iDSL language intuitive, presumably because of their familiarity with the service systems domain. Moreover, iDSL models were perceived to be remarkably concise and at the right level of abstraction. Also, the iDSL language suppports the separation of application and platform, as well as calibrations. System designers appreciated the level of

automation of the iDSL toolchain; after providing an iDSL model as input, visualized performance results for many designs are automatically returned. Under the hood, the so-called Modest language and toolchain are used to evaluate models using advanced model checking algorithms and discrete-event simulations. Modest has been selected because of its process algebra roots, its relatively readable language, and its support for multiple formalisms and ways of analysis.

For accurate results, iDSL uses a new evaluation technique based on probabilistic model checking, which yields full latency distributions. For this purpose, iDSL uses an algorithm on top of Modest, in which many iterations of Modest evaluations are performed in a systematic way. The evaluation technique exhaustively traverses the model, making the evaluation slow. iDSL counteracts this with automated model simplifications and making use of light-weight evaluation techniques initially that speed up the evaluation.

The applicability of iDSL on real systems has been assessed by conducting several case studies on *interventional X-ray systems* throughout the development of iDSL. Interventional X-ray systems are dependable medical systems that support minimally-invasive surgeries. Additionally, the wider applicability of iDSL has been assessed by applying iDSL on a load balancer case study.

In conclusion, iDSL delivers a fully automated performance evaluation chain from a high-level model to visualized performance results for service systems. This approach is not only very efficient, it also brings advanced formal performance evaluation techniques at the fingertips of system designers, without bothering them with the technical details of these.

Samenvatting

Een gïntegreerd systeem is een software-intensief systeem dat een bepaalde functie in een groter systeem vervult. Geïntegreerde systemen variëren van kleine draagbare apparaten, zoals digitale horloges en MP3 spelers, tot grote complexe systemen, zoals auto's en medische apparatuur. Ze zijn in de loop van de tijd steeds complexer geworden en moeten binnen een strikt budget gerealiseerd worden. Ze worden vaak gebruikt voor het uitvoeren van kritische taken, waardoor diens disfunctioneren kan leiden tot serieus letsel en zelfs een fatale afloop, zoals het geval is met medische systemen.

Een geïntegreerd systeem interageert op een dynamische manier met zijn omgeving, waardoor zijn *veiligheid* sterk van zijn zogenaamde presteren afhangt. Het presteren omvat de hoeveelheid werk die een systeem verzet in een gegeven tijd en wordt vaak gespecificeerd in termen van benuttingsgraden, wachtrijgroottes, doorvoersnelheden en reactietijden. Het is lastig een systeem goed te laten presteren, want: (i) de evaluatie van prestaties is vrijwel nooit een integraal onderdeel van het softwareontwikkelingsproces; (ii) systeemarchitecturen zijn in toenemende mate heterogeen, parallel en gedistribueerd; (iii) systemen worden vaak ontworpen voor verschillende productfamilies en configuraties; en (iv) metingen die inzicht geven in de systeemprestaties zijn vaak moeilijk te verkrijgen.

In dit proefschrift, beschouwen we zogenaamde *service(-georiënteerde) systemen*, een speciale klasse van geïntegreerde systemen. Een servicesysteem verleent services aan zijn omgeving en stelt die beschikbaar via zogenaamde serviceverzoeken. Het servicesysteem genereert precies één reactie op ieder verzoek. In een servicesysteem, zijn serviceverzoeken functioneel gescheiden, maar kunnen verzoeken wel elkaar's prestaties negatief beïnvloeden als ze tegelijktijdig gebruik willen maken van gedeelde middelen. Servicesystemen evalueren voordat deze volledige geïmplementeerd zijn is lastig; het beantwoorden van de vragen over de prestaties vereist modellen waarin tijdsaspecten en mechanismen om met onzekerheid om te gaan worden gecombineerd. Voorbeelden van onzekerheid zijn het gebruik van statistische voorspellingen voor executie-

tijden en systeemonderdelen die slechts ten dele in het model gespecificeerd zijn.

We hebben een nieuw raamwerk ontworpen voor de analyse van prestaties. De ontwerper definieert een hoog-niveau prestatiemodel, dat vervolgens wordt getransformeerd naar een onderliggend model. Evaluatie van dit onderliggende model leidt tot prestatieresultaten, die aan de systeemontwerper gepresenteerd kunnen worden.

De literatuur beschrijft vele zogenaamde gereedschapskisten die de analyse van prestaties voor de systeemontwerper vereenvoudigen. Een gereedschapskist omvat een verzameling van samenwerkende gereedschappen die (een gedeelte van) het proces van het analyseren van prestaties automatiseren. Echter, het merendeel van de gereedschapskisten maakt gebruik van formele evaluatietechnieken en verwacht dat de gebruiker over kennis van deze beschikt. Een systeemontwerper beschikt normaliter niet over deze kennis, omdat deze technieken niet tot zijn dagelijkse praktijk behoren. Verder leveren veel gereedschapskisten een model dat *niet* domein specifiek is, waardoor het modelleren veel tijd kost en het model lastig te begrijpen is. Daarnaast geven de meeste gereedschapskisten inaccurate resultaten, omdat voor het van genereren exacte resultaten meestal uitputtende evaluaties nodig zijn. Ten slotte komt het vaak voor dat de systeemontwerper veel tijd kwijt is aan het samenvoegen en visualiseren van de resultaten.

Het doel van dit proefschrift is om de bovenstaande tekortkomingen te niet te doen; de systeemontwerper moet in staat zijn een prestatiemodel te maken in de taal van zijn domein en resultaten in termen van dit domein te verkrijgen. Dit vereist een *hoog-niveau model* dat is toegespitst op zijn domein, dat bovendien bondig en expressief is. In het model moeten de applicatie (software) en platform (hardware) expliciet gescheiden zijn om onafhankelijke evaluaties van combinaties van deze te ondersteunen. Verder moet het mogelijk zijn een model te calibreren gebaseerd op metingen, teneinde het realistischer te maken Daarnaast moeten de resultaten op een visuele, begrijpbare wijze gepresenteerd worden voor een verscheidenheid van metrieken, zoals benuttingsgraden, wachtrijgroottes, doorvoersnelheden en reactietijden, om een beter inzicht te krijgen in de onderliggende karakteristieken van het systeem. Ten slotte moet de systeemontwerper in staat zijn om binnen een afzienbare tijd volledige geautomatiseerde evaluaties uit te voeren, voor vele ontwerpen en evaluatietechnieken.

Om het doel van het proefschrift te realiseren, hebben wij iDSL ontworpen en geïmplementeerd, een taal en gereedschapskist om de prestaties van servicesys-

temen te evalueren. Systeemontwerpers vonden de iDSL taal goed aansluiten bij hun domein en waardeerden de automatisering van de iDSL gereedschapskist, die automatisch visuele resultaten terug geeft. Achter de schermen maakt iDSL gebruik van de zogenaamde Modest taal en gereedschapskist om modellen te evalueren. Modest is gekozen vanwege zijn proces algebra achtergrond, de relatief leesbare taal, en ondersteuning voor verschillende formalismen en manieren van analyse.

Om exacte resultaten op te leveren, beschikt iDSL over een nieuwe evaluatietechniek gebaseerd op het herhaaldelijk toetsen van probabilistische modellen, die volledige distributies van reactietijden oplevert. iDSL realiseert dit met behulp van een algoritme boven op Modest, waarin meerdere Modest iteraties op systematische wijze uitgevoerd worden; Het model wordt op intensieve wijze geïnspecteert met als gevolg een trage evaluatie. Echter, iDSL versnelt de evaluatie door het model automatisch te versimpelen en door initieel lichtgewicht evaluatietechnieken toe te passen.

De toepasbaarheid van iDSL op echte systemen is vastgesteld door het uitvoeren van verschillende case studies op *interventionele röntgen systemen* gedurende het hele ontwikkeltraject van iDSL. Dit zijn medische systemen die minimaal invasieve chirurgische behandelingen ondersteunen. Tevens is de bredere toepasbaarheid van iDSL getoetst middels een taakverdeler casus.

Ten slotte, iDSL biedt een volledig geautomatiseerde keten voor de evaluatie van prestaties, van een hoog-niveau model tot visuele resultaten. Deze aanpak is niet alleen erg efficiënt, maar brengt bovendien formele technieken voor prestatieanalyse binnen handbereik van systeemontwerpers, zonder deze te confronteren met de technische details van deze.

Table of contents

CHAPTER 1

Introduction

1.1 Motivation

An embedded system is a computer system that has a dedicated function within a larger system, often with real-time computing constraints [134, 218]. In general, embedded systems are a heterogeneous combination of hardware and software. Today, the majority of the commonly used devices are embedded systems, e.g., about 98 percent of all microprocessors being manufactured are used in embedded systems [54]. Embedded systems range from portable devices such as digital watches and MP3 players, to large complex systems such as road vehicles [177], medical machines, and airplanes [122].

Embedded systems have faced a significant increase in complexity over time and are confronted with stringent cost constraints [38]. They are frequently used to perform safety critical tasks in challenging fields like aerospace, automotive and health-care. The malfunctioning of embedded systems can therefore lead to serious injury and fatalities, e.g., in case of medical systems [2, 135].

As embedded systems interact with their environments and these interactions are time critical, their safety is predominantly determined by their performance. Good performance is hard to achieve, because:

- Performance evaluation is often not an integral part of software engineering practices [163, 215].
- In early design stages, accurately predicting performance characteristics of real-time embedded systems is hard, since the real system does not exist yet [44, 91, 153, 181, 207].
- The architectures are increasingly heterogeneous, parallel and distributed [153, 160].

- Embedded systems are often designed for many product lines and different configurations, leading to a large variety of potential designs that each require performance evaluation [135, 160].
- Measurements to gain insight in the performance of an embedded system (also known as benchmarks) generally are expensive; they require the system to be built first and the system to be available for a long time to take sufficient measurements.
- Performance outliers, which are part of the important behaviour of embedded systems, are hard to detect, since they are often only rarely observed.

To narrow our scope, we only consider so-called *service-oriented systems* [102, 197], a special class of embedded systems that meet the following requirements:

- Service-oriented systems provide services to their environment, which are accessible via so-called service requests.
- Each of these service requests leads to exactly one service response, after some time.
- Individual service requests are isolated from each other in a service-oriented system. They do, therefore, not affect each other's functionality.
- Service requests may affect each other's performance as they potentially compete for shared resources in the service-oriented system.

Despite these restrictions, service-oriented systems can still be very complex. Namely, they are capable of handling many service requests in parallel, for different kinds of services [205]. To that end, they are equipped with multiple resources to process service requests with variable execution times.

To enable us to conduct our work on real service-oriented systems, we evaluate the performance of interventional X-ray (iXR, [158, 159]) systems, as explained in Chapter 3, in a number of case studies. iXR systems are dependable Medical Imaging Systems (MISs, [51, 164]), which are designed by our industrial partner Philips Healthcare as part of the COMMIT/Allegio Project [100]. Finally, we also evaluate the performance and energy consumption of load balancing strategies in data centers (as part of the STW project Cooperative Networked Systems [186]), to gain confidence that our performance evaluation approach can be widely applied, that is, also outside the scope of its development.

The remainder of this chapter is structured as follows. Section 1.2 first introduces the performance evaluation process. Section 1.3 then provides categories of indicators and summarizes the state-of-the-art of evaluation toolsets. Section 1.4 provides the goal of this thesis. Finally, Section 1.5 explains the structure of this thesis.

1.2 The performance evaluation process

Service-oriented systems are normally complex embedded systems, of which the safety is predominantly determined by their performance. To achieve good performance, it is indispensable to gain insight in this performance first, that is, prior to building the system. For this purpose we discuss two possibilities for a performance evaluation process, as follows. We first present performance evaluation based on *measurements* that are performed on one or more real service-oriented systems (in Section 1.2.1). Then we discuss performance evaluation based on measurements as well as predictions that are derived from a *performance model* of a service-oriented system (in Section 1.2.2).

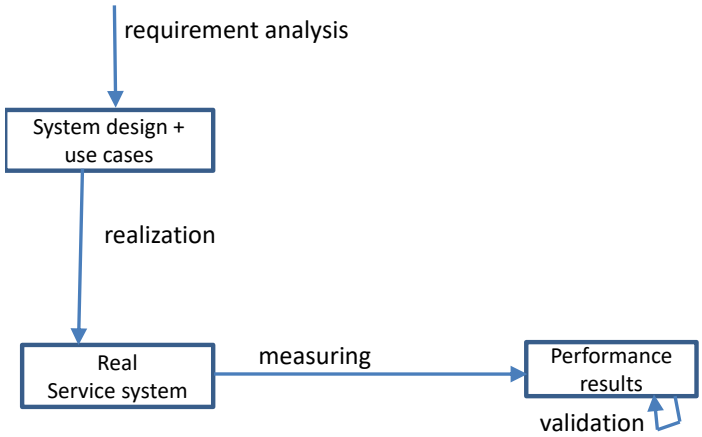
1.2.1 Performance measurement

Figure 1.1(a) shows the three components that make up system development including performance measurement, as follows:

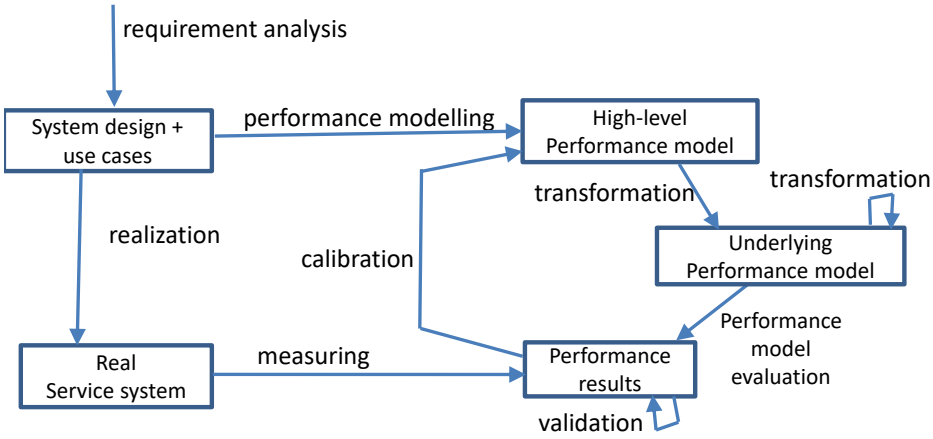
- System design + use cases: a blueprint of the system to be realized and how it is going to be used in its environment.
- Real service system: a realized service-oriented system.
- Performance results: outcomes for different performance properties, e.g., the average latency perceived for a service request, or the average queue size of a service.

The edges between the blocks in Figure 1.1(a) depict the steps of the performance measuring process. First, realization involves turning a system design into a real system. This normally includes putting the right hardware into place, as well as programming the software.

Second, measuring is about performing execution runs on the real system at which performance measurements are taken. In practice, measurements are often taken by adding stopwatches to the software code to record how long functions execute.



(a) System development including performance measurement



(b) System development including performance measurement and prediction

Figure 1.1: Two ways of applying performance evaluation.

Third, validation is concerned with creating confidence that the performance results are correct, e.g., if different measurement runs yield similar performance results, confidence in the correctness of the results increases.

Note that performance measurement normally requires a full realization of the system, including all its parts, viz., hardware and software parts. Hence, it cannot be applied when (i) some parts are still under development; (ii) the parts that are going to be used still need to be selected; and (iii) many designs are being developed at a time.

1.2.2 Performance measurement and prediction

Figure 1.1(b) shows the components that make up system development including performance measurement and prediction. This includes the components of Section 1.2.1, but, on top of that, the following two components:

- High-level performance model: a model of the system to be realized that abstracts as much as possible from performance-irrelevant aspects.
- Underlying performance model: a low-level model of the system that is derived from the high-level performance model. Due to its lower abstraction level, this model can generally be evaluated using generic tools and techniques.

The edges between the blocks in Figure 1.1(b) depict the steps of the performance prediction process. First, performance modeling involves creating an abstraction of the system design, use cases and results, yielding a performance model. This model is normally calibrated based on performance results, e.g., the execution time of a software component, that result from measuring on an yet existing (normally older) or partially implemented system. This is a reason why performance measurements (of Section 1.2.1) are still needed.

Second, the high-level performance model is transformed into an underlying performance model, which can be evaluated using more generic tools and techniques, such as discrete-event simulation [11,63] and analytical techniques, e.g., based on Queueing Networks (QNs, [10,70,86,92,114,196]). Note that it is not uncommon that multiple layers of transformations are performed to create the desired underlying performance model. For instance, in practice toolsets delegate work to other lower-level toolsets.

Third, the underlying performance model is evaluated using generic tools and techniques, and the results are generalized to be applicable to the high-level performance model.

Fourth, validation involves systematically comparing the performance outcomes for measurements (obtained as in Section 1.2.1) and predictions to gain confidence in future performance predictions made.

In this thesis, we will follow this process of Figure 1.1(b). That is, we will develop a performance evaluation methodology that uses measurements and that builds on other (low-level) techniques, to predict the performance of service-oriented systems.

1.3 The state-of-the-art

We present a number of indicators that allow us to determine the quality of performance evaluation toolsets. This allows us to see how well toolsets meet requirements that system designers generally value. After this, we apply them to a selected set of performance evaluation toolsets.

The indicators are in line with the prediction-specific components of the performance evaluation process (see Figure 1.1(b)) and are discussed in more detail in Appendix A. We summarize the indicators, as follows.

- $\mathcal{I}1$ High-level performance model, e.g., the ease, conciseness and freedom of modeling, and possibilities for calibration.
- $\mathcal{I}2$ Underlying performance model, e.g., the supported underlying models, and ease of transforming high-level models into underlying models.
- $\mathcal{I}3$ Performance evaluation, e.g., the supported evaluation techinques, the available tooling for automation and the execution time this takes, and support for Design Space Exploration (DSE).
- $\mathcal{I}4$ Performance results, e.g., the communicability, aggregation possibilities, visualization, and quality of the results.

In the following, the above indicators are used to systematically examine the current state-of-the-art regarding the performance evaluation process. For this purpose, we have selected fifteen performance evaluation toolsets that are well established in literature and implement the performance evaluation process based on measurements and predictions (of Section 1.2.2). After this, we have extensively evaluated these fifteen toolsets (as can be found in Appendix B), aggregated and summarized our findings, and categorized them by indicator, as follows.

$\mathcal{S}1$ High-level performance model (state-of-the-art regarding $\mathcal{I}1$):

Most currently available toolsets tend to provide a high-level performance model that contains a high level of detail. This makes the mapping to underlying performance models easy and makes the toolsets generically applicable.

However, these performance models are less suitable for communication and to model complex systems by system designers. Namely, they yield a large model in which implementation details and language constructs are often prominently visible. In several cases, toolsets do provide a graphical user interface (GUI) to compensate for this [59, 189].

The Y-chart philosophy [13, 14, 50, 119] is explicitly supported by a number of toolsets (e.g, see Appendix B.2, B.3, B.7, B.8, B.9 and B.14). Consequently, they allow the user to separately model an application (software) and a platform (hardware), which are glued together via a mapping. This makes it easy to replace either the application or platform by another instance (implementation) of it, e.g., to facilitate DSE (cf. Indicator $\mathcal{I}3$) or to run the application on an abstract platform (cf. Appendix C).

When measurements that have been performed on existing systems are available, these are in practice often used to calibrate the model, i.e., to slightly adjust the model to better describe reality. Most of the current toolsets do not explicitly support this, but do allow measurements to be added to their models manually, e.g., by using distributions provided by the language (cf. Appendix B.3, B.9 and B.10) or encoding measurements as distributions (cf. Appendix B.6).

$\mathcal{S}2$ Underlying performance model (state-of-the-art regarding $\mathcal{I}2$):

Most toolsets rely on one or more of the widespread underlying languages (cf. Section 2.1) under the hood. Many of these underlying languages are based on states and transitions between states, such as in Section 2.1.1 till 2.1.8.

The choice of underlying model(s) (cf. Section 2.3.2) depends, among others, on the presence (and separation) of discrete or continuous time, nondeterministic and/or probabilistic uncertainty, and clocks in the modeled system [89].

$\mathcal{S}3$ Performance model evaluation (state-of-the-art regarding $\mathcal{I}3$):

The underlying languages which are supported by a toolset determine which modes of analysis are supported (see Section 2.2). In general, toolsets tend to support simulations and only a few support model checking, such as the Modest toolset (cf. Appendix B.6) and UPPAAL (cf. Appendix B.15).

Ideally, toolsets automate the complete performance evaluation process. That is, ranging from high-level performance models to performance results. In practice, the system designer manually performs preprocessing, e.g., calibrating the system model using measurements, and postprocessing, e.g., visualizing performance results.

DSE is a way of finding attractive designs, i.e., parameter combinations, in the haystack of potential designs. Some toolsets support it, especially in combination with the Y-chart philosophy (cf. Indicator $\mathcal{I}1$). In other cases, DSE can either be explicitly modeled or accomplished by iteratively evaluating individual designs.

$\mathcal{S}4$ Performance results (state-of-the-art regarding $\mathcal{I}4$):

The supported modes of analysis mandate which performance results can be generated, such as utilizations, throughputs, latencies, and queue sizes. Latencies are supported most.

Several toolsets provide aggregated results, which are formed by combining existing results, e.g., computing the mean or median latency of a (preferably large) number of latencies.

A few toolsets provide visualizations and diagrams to make their results more meaningful and lively, e.g., Modular Performance Analysis (see Appendix B.7) and The Palladio framework (see Appendix B.9).

Finally, the quality of the performance results can be determined by assessing the accuracy, precision, and information [202] of the results (cf. Section 2.3.2). The quality depends on several factors, such as the quality of the high-level performance model, the supported underlying models, and the supported evaluation techniques (as graphically depicted in Figure A.1).

It can be seen that the state-of-the-art leaves room for improvement, which is discussed next (in Section 1.4).

1.4 The goal of this thesis

Section 1.3 summarizes the state-of-the-art of the currently available toolsets (of Appendix B). The toolsets are assessed on the basis of the prediction-specific components of the performance evaluation process (see Figure 1.1(b)). In the following, the requirements of the performance evaluation methodology of this thesis are discussed, which lead to concrete goals.

Driven by industrial use cases, our performance evaluation methodology should: (i) use as few costly measurements as possible; (ii) be able to evaluate a large number of complex designs; (iii) present its results intuitively via understandable (aggregated) metrics and visualizations; and, (iv) be applicable to real complex systems, i.e., provide high quality results within limited time and resources.

To achieve this, we would like the foreseen performance evaluation approach to perform relatively well on all indicators $\mathcal{I}1$ till $\mathcal{I}4$, instead of specializing in one of them per se, to provide a complete and automated approach from a high-level performance model to visualized results.

The goal of this thesis is then obtained by overcoming the shortcomings of the state-of-the-art in the following ways, categorized by indicator:

$\mathcal{G}1$ High-level performance model (goal of this thesis regarding $\mathcal{I}1$):

The provided high-level performance model should be expressive, yet concise. Therefore, the scope is limited to a special kind of embedded systems, the so-called service-oriented systems (as introduced in Section 1.1).

The language should be transformable into multiple formalisms to enable the use of different evaluation techniques.

An integrated Development Environment (IDE) and/or graphical user interface (GUI) should be provided that makes modeling easier, e.g., via syntax highlighting, intelligent code completion, and input validation.

As currently many toolsets do, the language supports the Y-chart philosophy by explicitly containing the application (software) and platform (hardware).

Also, the model should support calibration using both real measurements and predicted measurements that are based on existing measurements.

Finally, mechanisms to organize the language, such as compositionality, layers and hierarchies, and/or classes, are called for.

$\mathcal{G}2$ Underlying performance model (goal of this thesis regarding $\mathcal{I}2$):

The underlying performance model should be relatively abstract to easily model complex high-level performance models of service-oriented systems into it. This reduces the amount of manual, error-prone, labor and makes it easier to communicate the semantics of the high-level model. Moreover, in the worst case the complexity gap between both models can even become too large to make the transformation at all.

Also, the underlying performance model should be as simple as possible for quick analysis. Hence, it should support the enabling and disabling of properties like nondeterministic and probabilistic choices.

$\mathcal{G}3$ Performance model evaluation (goal of this thesis regarding $\mathcal{I}3$):

The whole performance evaluation process should be automated. That is, a system designer creates a model, which can be evaluated without any user interaction, and the results are automatically returned. Thus, DSE and post-processsing steps, such as rendering visualizations, are an integral part of the performance evaluation process.

Furthermore, for models of reasonable complexity, multiple modes of analysis should be supported. In addition to simulations for quick but less accurate results (that most current toolsets provide), model checking should be supported for accurate but generally slower results.

Finally, evaluating the model should take a limited amount of time and scale well for complex models to be used on real complex systems, such as real MISs.

$\mathcal{G}4$ Performance results (goal of this thesis regarding $\mathcal{I}4$):

Evaluations of a high-level performance model should lead to a range of results types, e.g., latency values, utilizations, latency bar charts, latency breakdown charts, absolute latency values, and latency distributions.

Whenever possible, the results should be presented in a visualization to increase the interpretability by a system designer.

1.5 The structure of this thesis

The goal of this thesis, as derived in Section 1.4, is accomplished via the following outline (as illustrated in Figure 1.2 and Table 1.1).

First, we provide background information (in Chapter 2) that supports Chapter 4, 5, 6, 7 and 8. It includes a literature study and two exploratory performance evaluation experiments.

Second, interventional X-ray systems, which are investigated via several case studies, are thoroughly described first in Chapter 3. They are used for different purposes, i.e., medical applications and configurations, and consist of two distinct subsystems at their highest level of hierarchy.

Third, four extensive experiments are performed on iXR systems, viz., to incrementally create iDSL, the performance evaluation language and toolbox of this thesis, in Chapter 4, 5, 6 and 7. Table 1.1 shows how these incremental steps contribute to the indicator classes. We separately provide a high-level performance model, underlying performance model, and performance evaluation and validation (cf. Figure 1.2) for each experiment.

Fourth, an additional experiment is performed on a load balancer to assess the broader applicability of iDSL (in Chapter 8). Load balancers are intrinsically different from iXR systems and also energy aspects are addressed besides performance ones. This indicates that iDSL can be more than just a performance tool.

Finally, this thesis contains three appendices. Appendix A conveys all the indicators used in the assessment of the performance evaluation process. Appendix B provides a comparison of several performance evaluation toolsets on the basis of the indicators defined in Appendix A. Appendix C contains a publication (as discussed in Section 2.5.1) in which performance evaluation is performed on a performance model that is automatically derived from an existing Domain Specific Language (DSL, [142, 208]) for medical machines.

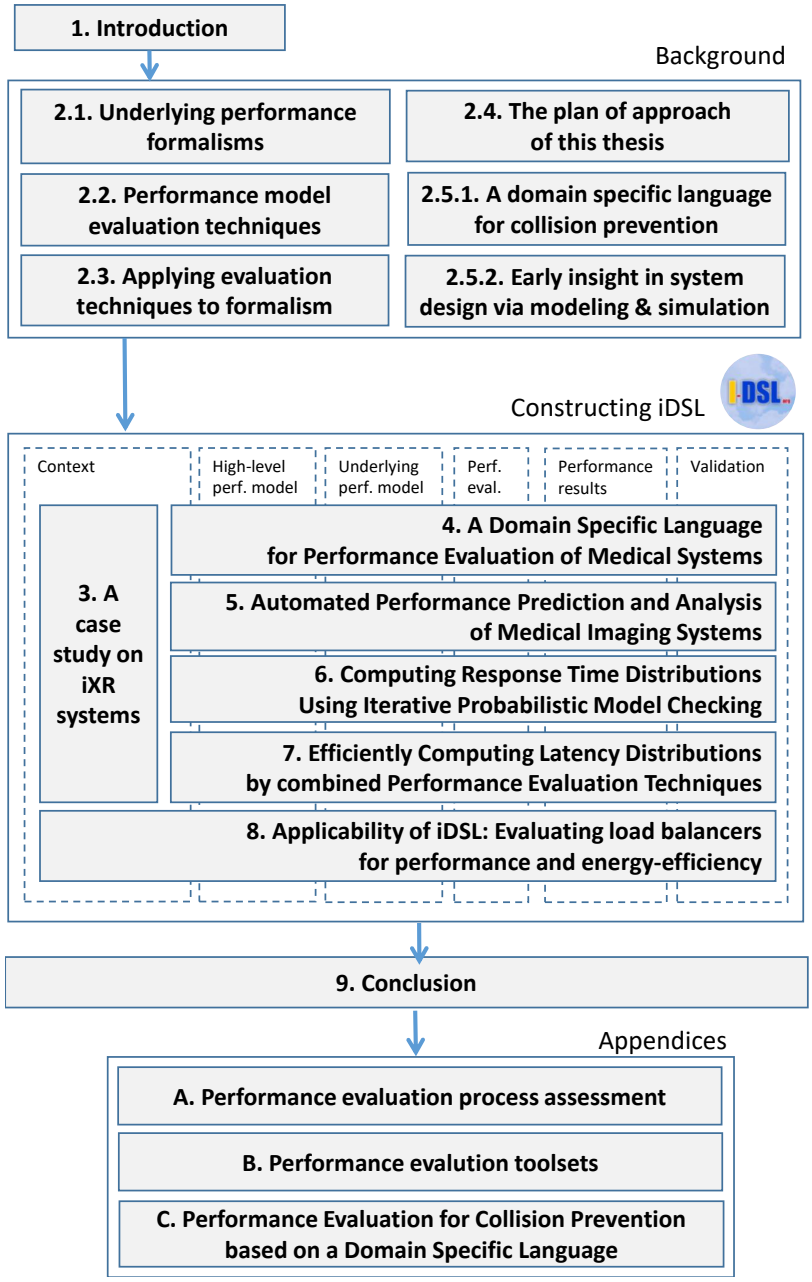


Figure 1.2: Thesis outline

Table 1.1: The incremental steps that constitute the realization of iDSL

Section	Context	High-level performance model	Underl. perf. model	Perf. model evaluation	Performance results
2.5.1	CPS	Add a transformation to a study DSL	POOSL	POOSL simulations	Latency distributions
2.5.2	IP	Build high-level POOSL libraries	POOSL	POOSL simulations	Latency distributions
4	IP	iDSL language	Modest	MODES simulations + UPPAAL	Lat. bar chart Lat. break-down chart Abs. latencies
5	IP	iDSL language + calibration + predict ecdfs	Modest	MODES simulations	Latency distributions
6	IP	iDSL language + model checking measure	Modest	Iterative MCSTA model checking	Latency distributions
7	IP	iDSL language + model checking measure 2.0 + automated model simplifications	Modest	Basic estimations +MODES Simulations +Iterative MCSTA model checking	Latency distributions
8	Load balancer	iDSL language + load balancer + energy-aware resource	Modest	MODES simulations	Energy-latency trade-offs

CPS=Cyber Physical System, a subsystem of an iXR system (cf. Section 3.1).
IP=Image Processing, a subsystem of an iXR system (cf. Section 3.2).
MODES=Modest Discrete Event Simulator [83, 144].
MCSTA=Model Checker of Stochastic Timed Automata [83].
POOSL=Parallel Object-Oriented Specification Language [59, 189].

CHAPTER 2

Background

This chapter provides background information that supports Chapter 4, 5, 6, 7 and 8, as follows.

First, a literature study is conducted that contains the available underlying performance models in Section 2.1, the available performance evaluation techniques in Section 2.2, and the application of these techniques to the formalisms in Section 2.3. Section 2.4 presents the plan of approach of this thesis.

Second, two exploratory performance evaluation experiments are conducted on Domain Specific Languages (DSLs, [142, 208]). In one experiment (cf. Section 2.5.1 and Appendix C), an existing DSL for collision prevention is automatically transformed into a performance model to evaluate the performance for different scenarios. In another experiment (cf. Section 2.5.2), high-level components are created for Image Processing (IP, [183]) using Parallel Object-oriented Specification Language (POOSL, [59,69,209]). These components can be used to simplify DSL transformations and thereby reduce the amount of work and errors. Additionally, they make the semantics of the high-level language easier to understand.

2.1 Underlying performance model formalisms

In this section, we present, without attempting to be exhaustive, a variety of formalisms that can be used as a representation for the underlying performance model (cf. Indicator $\mathcal{I}2$ of Appendix A), as follows. In Section 2.1.1 till 2.1.6, we discuss transition systems, ranging from Labelled Transition Systems (LTSs, [195]) to Stochastic Timed Automata (STA, [32,82]). After this, two categories of transition systems are shed a light on, namely Markovian (in Section 2.1.7) and Hybrid (in Section 2.1.8) transition systems. We conclude with addressing four formalisms that are not transition systems (in Section 2.1.9)

For categorizing underlying performance models that are based on transition systems (in Section 2.1.1 till 2.1.8), the following four dimensions are used [89].

1 *Probabilism* involves jumping from one state to a target state, according to a probability distribution over target states. This means that whenever a transition from a starting state to a target state is made, it is uncertain what this target state will be. For instance, a coin-flip could be used to uniformly select a target state from two candidate states. In a model, transitions leads to either: (i) a fixed target state with probability 1 (*no* probabilism); (ii) one out of a countable infinitely number of states (*discrete* probabilism); or, (iii) one out of an uncountably infinite number of states (*continuous* probabilism).

2 *Nondeterminism* is an unquantified choice between two or more alternatives, which can been seen as the previous probabilism dimension but without knowing the probabilities. In a model, either: (i) all transitions with a given starting state and alphabet letter lead to a fixed target state (*no* nondeterminism); (ii) all transitions have finite or countably infinite number of alternatives per nondeterministic choice (*discrete* nondeterminism); or, (iii) all transitions have uncountably infinite number of alternatives per nondeterministic choice (*continuous* nondeterminism).

3 *Stochastic residence times* are about the distribution of the amount of time spend before jumping from one state to the next state. For instance, after making a transition to a given state, it takes between 10 and 20 time units, uniformly distributed, before the next transition is made. A model either does not support residence times (*no* stochastic residence times), supports residence times adhering to an exponential probability distribution (*exp.* stochastic residence times), or supports continuous distributions (*general* stochastic residence times).

4 *Flow* is concerned with the deterministic evolution of continuous values over time, such as clocks that continuously increase at a constant rate of 1. A model either has no clock (*no* flows), only has clocks that increase at a constant rate of 1 (*clock* flows), or has clocks with general evolutions (*general* flows).

2.1.1 Labeled Transition Systems

A Labelled Transition System (LTS, [195]) consists of a set of states, which are connected in a directed way by transitions. Transitions are in turn labeled with

elements from a given alphabet, e.g., A ,B ,C ,.... An LTS has a starting state, from which it evolves into subsequent states.

An LTS jumps to a new state when (i) a transition exists from the current to the new state, and (ii) the label of this transition matches an externally given input label. When multiple transitions with the same label can be applied, nondeterminism occurs, i.e., the LTS can jump from one state to one out of multiple states. In this case, it is not defined what the next state will exactly be.

Hence, an LTS mainly offers nondeterminism but also supports models without nondeterminism, i.e., when each transition with a given starting state and given alphabet symbol leads to a unique new state. Hence, both the values *discrete* and *no* are valid for nondeterminism (as can be seen in Table 2.1).

Table 2.1: The support of an LTS for four dimensions

Probabilism	**Nondeterminism**	**Flows**	**Stochastic residence times**
no	discrete no	no	no

Note that an LTS supports *no* on the nondeterminism, which does not imply by any means that an LTS does not support nondeterminism. Instead, it means that an LTS can be free of nondeterminism, as just mentioned.

For illustration, Figure 2.1 depicts an LTS with starting state s_0 and follow up states s_1 (with symbol A) and s_2 (with symbol B). This LTS is deterministic, because each transition from state s_0 has a unique element from the alphabet.

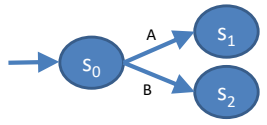


Figure 2.1: the underlying formalism LTS

Example: An LTS can be used to answer reachability questions, e.g., is it, at all, possible that (part of) my system will end up in a livelock or deadlock? This can be performance-relevant since livelocks and deadlocks can lead to a non-responsive system with arbitrarily high or even (theoretically) infinite service latencies.

2.1.2 Discrete-time Markov Chains

Like an LTS, a Discrete-Time Markov Chain (DTMC) also consists of states, which are connected by transitions. Unlike an LTS, transitions are free from labeled elements from a given alphabet. Instead, they are augmented with probabilities $p \in [0 : 1]$, the likelihood that a transition is selected, and thus sum up to 1 for each originating state.

Therefore, when a state has transitions to multiple states, a probabilistic choice is made among these states. Probabilism, or discrete jump target distributions, is the main feature DTMCs provide (see Table 2.2).

Table 2.2: The support of a DTMC for four dimensions

Probabilism	**Nondeterminism**	**Flows**	**Stochastic residence times**
discrete no	no	no	no

Note that probabilism is optional for a DTMC (viz., it has values *discrete* and *no*) when all transitions have a probability equal to 1, in a similar way nondeterminism is optional for an LTS (in Section 2.1.1).

For illustration, Figure 2.2 shows a DTMC with starting state s_0. The likelihood that s_1 will be the next state is 0.6, and for s_2 it is 0.4.

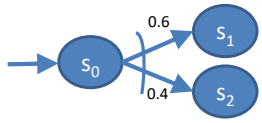


Figure 2.2: the underlying performance formalism DTMC

Example: A DTMC can be used to compute steady state distributions, i.e., for each state, the probability that the DTMC is in this state after a large number of transitions. These probabilities can be used to answer a variety of performance questions. For instance, at any time, what is the probability that a queue size is greater than a given value? This information might be valuable to identify bottlenecks.

2.1.3 Timed Automata

Timed Automata (TA, [20, 104]) also consist of states and transitions. Transitions are labeled with elements from a given alphabet (like an LTS) and with a number of clocks. Hence, a TA is an LTS extended with clocks (see Table 2.3).

Clocks are real variables that continuously increase over time with rate 1. Besides labels, transitions of a TA contain assignments in which a clock can be reset to 0 and guards that are invariants (boolean expressions) on clocks. Guards have to be true for the transition to take place. Using clocks and guards, TA enable continuous nondeterminism, viz., nondeterministic time on continuous intervals defined using guards.

Table 2.3: The support of a TA for four dimensions

Probabilism	**Nondeterminism**	**Flows**	**Stochastic residence times**
no	continous discrete no	clocks no	no

For illustration, Figure 2.3 shows a TA with starting state s_0 and follow up states s_1 (with symbol A and a reset of clock c) and s_2 (with symbol B). State s_1 has, in turn, transitions to states s_3 and s_4 (both with symbol A). States s_3 and s_4 have different timing constraints, viz., either within or after 15 time units, respectively. The selection of a time in state s_1 is continuous nondeterministic, since the ranges $[0 : 15]$ and $(15 : \infty)$ are of uncountably infinite size.

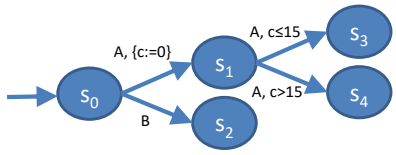


Figure 2.3: the underlying performance formalism TA

Example: TA are convenient for checking hard real-time constraints, e.g., is the response time of my system, in all conceivable scenarios, less than 20 milliseconds?

2.1.4 Probabilistic Automata / Markov Decision Process

A Markov Decision Process (MDP, [29]), also known as an probabilistic automaton (PA), again consist of states and transitions. Transitions are alternating nondeterministic and probabilistic choices.

Therefore, an MDP combines the advantages of LTSs (of Section 2.1.1) and DTMCs (of Section 2.1.2), which gives an MDP both probabilistic and nondeterministic capabilities (see Table 2.4).

Table 2.4: The support of a PA/MDP for four dimensions

Probabilism	**Nondeterminism**	**Flows**	**Stochastic residence times**
discrete no	discrete no	no	no

For illustration, Figure 2.4 shows an MDP in which nondeterministic and probabilistic decisions are alternated. Starting state s_0 leads to either state $s_{0,a}$ (via symbol A) or state $s_{0,b}$ (via symbol B). From state $s_{0,a}$, s_1 (with probability 0.6) and s_2 (with probability 0.4) can be reached. State $s_{0,b}$ surely (with probability 1) leads to state s_3. States $s_{0,a}$ and $s_{0,b}$ (in red) are states that are left as soon as they are entered, also known as vanishing states, because each nondeterministic choice requires a probabilistic choice to be performed immediately as well. These vanishing states are not part of the true state space.

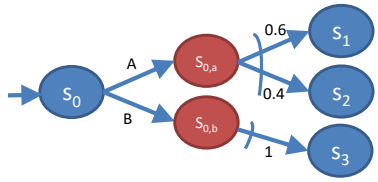


Figure 2.4: the underlying performance formalism PA/MDP

Example: An MDP can conveniently be used to answer statistical questions of a nondeterministic system. Nondeterminism may come from, among others, underspecification and parallelism. For instance, with what probability is each response time in my system, regardless of what scheduler is used, below 20 milli-seconds?

2.1.5 Probabilistic Timed Automata

A Probabilistic Timed Automata (PTA, [15, 129]) consist of states and transitions between states. Each transition is either a nondeterministic choice with a number of clocks, or a probabilistic choice.

Hence, a PTA can be seen as TA (of Section 2.1.3) extended with probabilities or, similarly, as an MDP (of Section 2.1.4) extended with clocks. As a result, a PTA supports a combination of nondeterministic, probabilistic and clock capabilities (as illustrated by Table 2.5).

Table 2.5: The support of a PTA for four dimensions

Probabilism	**Nondeterminism**	**Flows**	**Stochastic residence times**
discrete no	continous discrete no	clocks no	general exp. no

For illustration, Figure 2.5 depicts a PTA that is similar to the previously shown PA/MDP (in Figure 2.4) except for two differences.

First, the transition between state s_0 and vanishing state $s_{0,b}$ is different from Figure 2.4. Namely, it has a reset for clock c.

Second, vanishing state $s_{o,b}$ has two outgoing transitions of (both with symbol A) with timing constraints instead of one outgoing transition in Figure 2.4. These timing constraints are within or after 15 time units, inspired by the TA of Figure 2.3.

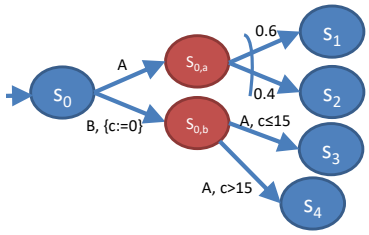


Figure 2.5: the underlying performance formalism PTA

Example: As with an MDP, a PTA can also be used to answer statistical questions of a nondeterministic system. Besides this, a PTA supports clocks, which can be used to model nondeterministic time. This leads to a range of possible answers, e.g., with what probability range is each response time in my system, regardless of what scheduler is used, below 20 milli-seconds?

2.1.6 Stochastic Timed Automata

Stochastic Timed Automata (STA, [32, 82]) consist of states and transitions. These transitions are alternating nondeterministic and probabilistic choices. The nondeterministic choices contain a symbol of the alphabet. They also contain a so-called "*when* boolean condition", i.e., a transition may take place when this boolean is true, and a so-called "*urgent* boolean condition", i.e., a transition must take immediately when this boolean is true. Combined, they can be used to mark a time interval on which the transition has to take place.

Using these conditions, a STA provides, in addition to a PTA, support for continuous probability distributions, and general stochastic residence times, e.g., a residence time defined by a range of numbers (as can be seen in Table 2.6).

Table 2.6: The support of a STA for four dimensions

Probabilism	**Nondeterminism**	**Flows**	**Stochastic residence times**
continuous discrete no	continuous discrete no	clocks no	general exp. no

For illustration, Figure 2.6 depicts an STA with a structure that is similar to the PA/MDP of Figure 2.4. The transition from state s_0 to vanishing state $s_{0,a}$ contains alphabet symbol A. The when condition is $c \geq 5$, and the urgent is $c \geq 10$. The combination of conditions make the transition fire after t time units, with $t \in [5 : 10]$. This is a uncountably infinite range and thus implies continuous nondeterminism over time. Finally, the transitions from vanishing state $s_{0,a}$ to states s_1 and s_2 are probabilistic choices (as with PA/MDPs) with respective probabilities 0.6 and 0.4.

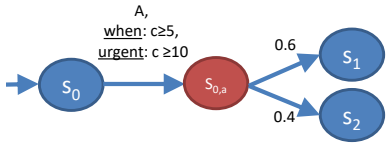


Figure 2.6: the underlying performance formalism STA

2.1.7 Markovian transition systems

Markovian transition systems are transition systems that adhere to the so-called memoryless property, i.e., having stochastic residence times (e.g., execution times of components) that are negative exponential. Table 2.7 displays the properties of three Markovian transitions systems, namely a Continuous-Time Markov Chain (CTMC), an IMC (interactive Markov chain) and an MA (Markov automaton), which are discussed next.

CTMC A CTMC [7, 92] is similar to a DTMC but makes use of continuous time instead of discrete time. A CTMC consists of a number of states and a so-called transitions rate matrix, which describes the negative exponential rate of residing in one state before moving to another state. A transition from the current state to a new state, including the residence time in this current state, is determined by sampling all the potential new states using the respective rates of the transition rate matrix.

IMC An IMC [96] can be seen as a CTMC extended with nondeterminism (see Table 2.7) or, similarly, as an LTS (of Section 2.1.1) extended with exponentially delayed transitions. An IMC can be defined as a combination of: Markovian states that behave like CTMC states and have outgoing Markovian transitions with rates, and interactive states that behave like LTS states and have outgoing interactive transitions with symbols of the alphabet. Both Markovian and interactive transitions can lead to either Markovian or interactive states.

MA An MA [60, 81] can been seen as a continuous time version of a PA (of Section 2.1.4). Its transitions are either an action, which consist of a letter of the alphabet and a discrete probability distribution over states, or a negative

Table 2.7: Properties supported by Markovian transition systems

Underlying model formalism	**Proba-bilism**	**Nondeter-minism**	**Flows**	**Stochastic residence times**
CTMC	no	no	no	**exp.**
IMC	no	discrete no	no	**exp.**
Markov automata	discrete no	discrete no	no	**exp.**

exponential rate. Consequently, an MA can be used to model a combination of action transitions as with an LTS (of Section 2.1.1), probabilistic branching of a DTMC (of Section 2.1.2), and exponential delays of a CTMC.

The memoryless property of the just discussed Markovian systems can make analysis easier, but limits the set of service-oriented systems that can be modeled by them to a great extend. Namely, in practice, execution times of services often do not follow a negative exponential distribution but, for instance, an arbitrary stochastic distribution instead. Therefore, Markovian transition systems are not further addressed in this thesis.

2.1.8 Hybrid transition systems

Hybrid transition systems are transition systems that support general flows in addition to clocks, which makes them very suitable for modeling systems that exhibit both continuous and discrete dynamic behavior, e.g., the temperature of a room (continous) that is regulated by an electronic thermostat (discrete).

Table 2.8 presents the properties of three Hybrid transition systems, namely a Hybrid automaton (HA), a Probabilistic Hybrid automaton (PHA), and a Stochastic Hybrid automaton (SHA), as follows.

HA An HA [94] is a finite state machine with a finite set of continuous variables whose values are described by a set of ordinary differential equations (ODEs). An HA can therefore be seen as a TA (of Section 2.1.3) extended with continuous variables instead of clocks.

Table 2.8: Properties supported by Hybrid transition systems

Underlying model formalism	**Proba-bilism**	**Nondeter-minism**	**Flows**	**Stochastic residence times**
Hybrid automata	no	continous discrete no	**general** clocks no	no
Prob. Hybrid automata	discrete no	continous discrete no	**general** clocks no	no
Stoch. Hybrid automata	continous discrete no	continous discrete no	**general** clocks no	general exp. no

PHA A PHA can be seen as a HA extended with probabilities. A PHA can thus be used to model systems with both probabilistic traits and complex dynamics.

SHA A SHA [101] can be seen as a HA extended with stochastic behavior, or an STA (of Section 2.6) extended with continuous dynamics. Consequently, an SHA combines nondeterministic choices, continuous system dynamics, stochastic decisions and timing, as well as real-time behaviour.

Hybrid systems, such as HAs, PHAs and SHAs, can be model checked, enabling nondeterminism, e.g., using the prohver tool of the Modest toolset [67]. They can also be simulated, e.g., using the Anylogic tool [25]. Nevertheless, by supporting flows, hybrid systems are harder to analyze and less scalable, while the systems we would like to support do not require flows. Hence, hybrid transition systems are not further addressed in this thesis.

2.1.9 Other underlying performance model formalisms

We discuss four kinds of formalisms that are different from transition systems, namely back-of-the-envelope models, Queuing networks, Petri-nets, and data-flows.

Back-of-the-envelope models describe a system in a simple way using mathematical concepts and language. They are more than a guess but less than an accurate calculation or mathematical proof. They are common practice in natural sciences (e.g., physics), social sciences (e.g., economics), as well as in engineering disciplines as addressed in this thesis.

Back-of-the-envelope models can be, among others, a set of ordinary differential equations (ODE, [210]), simply one arithmetic expression [202], or even a full-fletched spreadsheet [3].

Queueing networks (QN, [10, 70, 86, 92, 114, 196]) are networks of queues in which a number of queues are connected by customer routings. Queueing theory then forms the mathematical study of QN. QNs can be evaluated in different ways [70, 92]. They are frequently transformed into CTMCs, but can also be transformed into DTMCs (of Section 2.1.2) after which evaluation techniques for transition systems (as discussed in Section 2.2) can be applied to them. For specific types of queueing networks, efficient analysis is feasible, e.g., queueing analysis of pools in soft real-time system to compute mean queue lengths [114], and so-called product form queuing networks that have a simple closed form expression of the stationary state distribution that allow to define efficient algorithms for obtaining average performance measures [10].

A Layered Queueing Network [66, 124, 196] is Queueing Network extension where the service time for each job at each service node is given by the response time of a Queueing Network. On top of that, the service time of the latter Queueing Network may again be determined by further nested networks.

Petri-nets [72, 170] describe distributed systems using places, transitions and arcs, and can be mathematically represented as a bipartite graph. Petri-nets are either discrete, continuous or hybrid, and related to discrete, continuous or hybrid automata (of Section 2.1.1 till 2.1.8), respectively.

Like QNs, Stochastic Petri-nets [114] can be transformed into CTMCs and DTMCs (of Section 2.1.2), which thereby enable the application of certain evaluation techniques for transition systems (as discussed in Section 2.2). Additionally, under specific (pretty limited) conditions, a more efficient analysis can be performed on Petri-nets directly, e.g., efficiently determining whether certain states of the Petri-net are reachable [127]. Compared to transition systems, Petri-nets provide a somewhat higher-level formalism.

Many extensions of Petri-nets exist, such as Coloured Petri-nets [108, 169] in which every token has a value to distinguish tokens, as well as Stochastic

Petri-nets [1], where transitions fire after a probabilistic delay determined by a random value.

Dataflows come in many flavors of which Scenario Aware DataFlow (SADF, [188]) and Synchronous DataFlow For Free (SDF3, [185]) are widespread. Dataflow models consist of actors that are connected via channels on which tokens flow with a certain rate. Dataflow models are popular because of their simplicity, which makes them, easy to analyze. As a consequence, they lack expressibility, e.g., the Markov transition systems (of Section 2.1.7).

2.2 Performance model evaluation techniques

In this section, performance model evaluation techniques for underlying performance models are introduced (cf. Indicator $\mathcal{I}3$ of Appendix A). Once an (underlying) performance model (of Section 2.1) has been selected, a number of performance model evaluation techniques can be applied to it. We discuss back-of-the-envelope analysis (in Section 2.2.1), discrete-event simulation (in Section 2.2.2), traditional model checking (in Section 2.2.3), and probabilistic model checking (in Section 2.2.4). For each evaluation technique, we provide a brief introduction, its applicability, the way it performs analysis on a model, and the results it delivers.

2.2.1 Back-of-the-envelope analysis

Once a high-level performance model is transformed into a back-of-the-envelope model, the rules of mathematics are applied to it to yield a solution.

For instance, Figure 2.7 provides an equation for estimating the average waiting time of a costumer for a single server queueing station with unlimited buffer capacity and unlimited customer population, based on a so-called $M|G|1$ queueing model [92, Chapter 5]. The equation shows that in order to estimate the average waiting time (on the left-hand side), the arrival rate (λ), second moment of the service time distribution ($E[s^2]$), and utilization (ρ) need to be known (on the right-hand side). Once these three values are known, solving the equation is simply a matter of evaluating an arithmetic expression.

For another example, to solve ordinary differential equations (ODEs), an algorithm might include steps like separating variables and finding an integral for an exact equation. These steps are normally performed (relatively) quickly by a computer.

$$E[w] = \frac{\lambda E[s^2]}{2(1-\rho)}$$

Figure 2.7: Back-of-the-envelope analysis.

Applicability Back-of-the-envelope models can be very expressive, but there are no guidelines about obtaining a back-of-the-envelope model from a high-level performance model. In general, it might also be hard to express the system dynamics that, e.g., parallel processing and scheduling bring about [13, 14].

Model analysis By definition, a back-of-the-envelope calculation is a rough calculation, typically written down on any available piece of paper, e.g., the actual back of an envelope. Hence, back-of-the-envelope models can be evaluated quickly, because they generally comprise simple mathematical constructs that computers can compute quickly.

Results Back-of-the-envelope analysis, often leads to an underlying performance model that highlights only one aspect, e.g., an arithmetic expression to compute a service latency [202], leading to a low variety of results. In general, the quality of the results is low, due to limited expressibility resulting from the desire of keeping the model small and simple.

2.2.2 Discrete-event simulation

Discrete-event simulation is an evaluation technique in which a model is traversed in a random way, i.e., nondeterministic or probabilistic choices are resolved using a random number generator [143]. It is therefore a powerful technique to get a first impression, viz., an approximation, about complex models.

Occasionally, discrete-event simulation is also referred to as statistical model checking. However, they are not necessarily synonymous. For instance, [136] states that a statistical model checking algorithm comprises the performance of multiple discrete-event simulations under the hood.

Applicability Table 2.9 shows that discrete-event simulation supports many values for each of the four dimensions (of Section 2.1), e.g., discrete-event simulation supports *continuous*, *discrete* and *no* probabilism. In effect, discrete-event simulation can be applied to many underlying formalisms, such as an LTS, DTMC, PA/MDF, TA, PTA, and STA.

Table 2.9: The supported properties of discrete-event simulation

Probabilism	**Non-determinism**	**Flows**	**Stochastic residence times**
continous discrete no	continous (resolution needed) discrete (resolution needed) no	clocks no	general exp. no

However, Table 2.9 also conveys that ways to resolve nondeterminism are needed, as follows. When a nondeterministic choice with a finite number of options occurs during analysis, i.e., discrete nondeterminism, it is common for a evaluation time scheduler to turn it into a uniform probabilistic choice. In case of continuous nondeterminism, e.g., a transition must take place between 5 and 10 time units, as soon as possible (ASAP) scheduling is a common way to resolve it. Resolving nondeterminism inherently yields results that are of reduced quality, viz. a uniform choice leads to average behavior while the underlying distribution might be very different, and ASAP scheduling neglects much of the state space by only considering one out of many options, respectively.

Model analysis Figure 2.8 shows an underlying performance model that has been derived from a high-level performance model. Using discrete-event simulation, a random path is taken through the state space, namely the numbered, red states. The numbers of the states indicate the order in which they are examined. Note that state 1 and 2 have multiple outgoing transitions in which case only one of these outgoing states is selected for further analysis. Although this way of analysis conveys information about the state space quickly, it is also inherently inaccurate, viz., the blue states in Figure 2.8 are not examined.

However, depending on various properties, such as the length and number

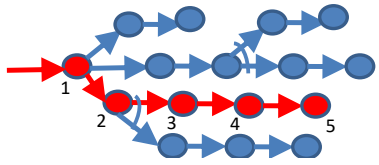


Figure 2.8: Discrete-event simulation, graphically depicted.

of individual discrete-event simulations performed, the size of the state space, and the way similar states are dispersed on this state space, statistics can and have been used to assure the quality of results [92, Chapter 18], as follows. First, many states are visited using a number of simulations in which something is measured, e.g., a latency. Second, some measurements are used to compute mean values, e.g., an average latency, and confidence intervals, e.g., a latency is on interval $[8 : 10]$ with probability ≥ 0.95.

Additionally, rare-event simulation [176, 180] puts emphasis on discovering rare events or states via simulations, e.g., by saving states that lead to a rare event and using them as the starting point of new simulation sub-runs. Consequently, simulations become more efficient, because all possible states are discovered sooner.

Results Due to the fast analysis that discrete-event simulation enables, it can be applied to STA models and its various sub-models. Therefore, it can be used to return a great variety of derivable results. However, the quality of these results (cf. Section 2.3.2) is not always optimal, because examining only part of the state space comes at the cost of a reduced accuracy and precision that cannot always be compensated for by statistics or rare event simulation.

2.2.3 Traditional model checking

Traditional model checking, is an exhaustive check of properties of all reachable states. Therefore, the set of states sometimes has to be finite to ensure computability [171]. In practice, it can only be applied to fairly simple model because of the so-called state space explosion problem [36, 37], viz., the number of states grows too large to be evaluated. This number of states can easily grow large, since it grows exponentially as the number of state variables in the model increases.

Applicability Table 2.10 shows that traditional model checking does not support probabilism, neither continuous nor discrete. Hence, it can only be applied to the underlying formalisms LTS (of Section 2.1.1) and TA (of Section 2.1.3).

Model analysis Figure 2.9 depicts a sample evaluation using a depth-first search, although breath-first searches are common practice as well [19, 132]. Again, the numbers of the states indicate the order in which they are examined.

Table 2.10: The supported properties of traditional model checking

Probabilism	**Non-determinism**	**Flows**	**Stochastic residence times**
no	continous (time) discrete no	clocks no	no

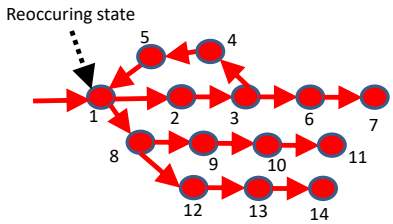


Figure 2.9: Traditional model checking, graphically depicted.

Opposed to a statistical model checking evaluation, all reachable states are examined. When a state has multiple outgoing transitions, such as with states 1, 3 and 8, their sub-models are examined one by one. A local search ends when a state is reached that has no outgoing states or when a state is reached that has already been visited, e.g., the transition from state 5 to state 1 that has already been visited.

Results Traditional model checking can only be used to retrieve a limited variety of results, because it cannot handle probabilism and suffers from the state space explosion problem. The quality of the results are good, because examining the whole reachable state space guarantees precise and accurate results.

2.2.4 Probabilistic Model checking

Probabilistic model checking extends traditional model checking with probabilistic features, leading to a challenging mixture of nondeterministic and probabilistic traits. Even more than before, the state space explosion problem [36,37] is relevant here.

Table 2.11: The supported properties of probabilistic model checking

Probabilism	Non-determinism	Flows	Stochastic residence times
discrete no	continous (time) discrete no	clocks no	no

Applicability Using Table 2.11, it can be deducted that probabilistic model checking can be applied to underlying formalisms LTS, TA and PTA. On the downside, probabilistic model checking can, for instance, not be applied to an STA, due to a lack of support for continuous probabilism.

Model analysis Figure 2.10 illustrates how probabilistic model checking is put into practice. Besides addressing which states are reachable as with traditional model checking, their respective probabilities are administered too. To accommodate this, a breath-first search is commonly employed. When all reoccurring states have been found, the true probabilities values are approximated using so-called value iteration, as follows.

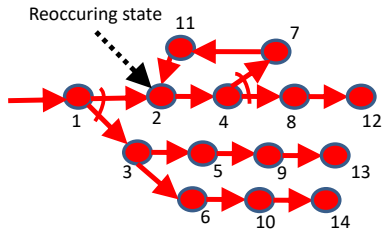


Figure 2.10: Probabilistic model checking, graphically depicted.

Value iteration [29] is a well-known process used to compute an optimal policy and value of an MDP (of Section 2.4). Let S be the set of all states and $S_t \subset S$ be the set of states for which a property under investigation holds, e.g., a service request has led to a response within 20 time units. Next, we investigate, for all states that are not in S_t, the probability that they will lead to a state in S_t. Let $V_k : S \rightarrow [0 : 1]$ is the computed probability that one of set of states S_t

is reached from a given state $s \in S$, after k iterations. Let $V_0(s) = 1$ for $s \in S_t$, and $V_0(s) = 0$ for $s \notin S_t$. Next, V_{k+1} is computed based on V_k, for each $s \in S$ and $k = 0, 1, 2, \ldots$,n, as follows:

$$V_{k+1}(s) = \max_{t \in tr(s)} \sum_{s' \in target(t)} prob(t, s') \cdot V_k(s') \tag{2.1}$$

where n is the number of transitions, $tr(s)$ the set of transitions from state s, $target(t)$ the set of target states of transition t, and $prob(t, s')$ the probability that transition t leads to state s'.

The number of iterations n has to be carefully selected and may depend on, among others, the state space size and structure. As a rule of thumb, more iterations lead to better results but require more computation time. Equation 2.1 yields a maximum probability. The minimum probability is obtained analogously. If the minimum and maximum probability outcomes are different, this is due to nondeterminism in the model.

Results Probabilistic Model checking can be used to retrieve a reasonable variety of results, due to the powerful combination of nondeterminism, probabilism and clocks. The quality of these results are reasonably good, because again the whole reachable state space is examined.

2.3 Applying evaluation techniques to formalisms

In this section, we illustrate how various evaluation techniques (of Section 2.2) can be applied to underlying formalisms (of Section 2.1), as follows. In Section 2.3.1, we convey which underlying formalisms can be applied, given an evaluation technique. In Section 2.3.2, we discuss three criteria for selecting a suitable combination of an evaluation technique and an underlying formalism.

2.3.1 The applicability of evaluation techniques to formalisms

In practice, a formalism to model a system in (of Section 2.1) is selected first, followed by an evaluation technique (of Section 2.2) that can be applied to this formalism. Table 2.12 shows, for six different underlying formalisms, which of four evaluation techniques they support, as follows.

First, back-of-the-envelope analysis is a broad concept, which generally means a rough calculation. Therefore, it can, in principle, be applied to all underlying

formalisms provided that the measure(s) of interest can be derived from the formalism.

Second, discrete-event simulation is in principle applicable to all formalisms, but requires nondeterminism, if present, to be resolved (as explained in Section 2.2.2, application). In case of discrete nondeterminism, nondeterminism can be resolved by configuring the scheduler to randomly select one of the options. In case of continuous nondeterminism, nondeterminism can be resolved by making the scheduler select either the minimum, maximum or average value of the interval of options, e.g., an as soon as possible scheduler for time. In Section 8.2.1, we resolve discrete nondeterminism in the model by adding a random (cf. Equation 8.4) number in the range $[0, 1)$ to a so-called load balancing policy. This prevents the scheduler to be confronted with nondeterminism.

Third, traditional model checking can be applied to all formalisms but STA. However, in case of a PTA, stochastic residence times need to be represented as alternatives with delays in the model. Moreover, it requires probabilistic alternatives to be replaced by nondeterministic alternatives in the model, for a DTMC, MDP and PTA.

Fourth, the applicability of probabilistic model checking is similar to traditional model checking, but only requires stochastic residence times to be represented as alternatives with delays in case of a PTA.

2.3.2 Criteria for evaluation techniques and formalisms

In addition to knowing which underlying formalism and evaluation technique can be used together, the system designer needs to have insight in the consequences of selecting a specific combination of formalism and evaluation technique. For this purpose, we address three criteria that form a trade-off, i.e., a gain on one criterion involves a loss on another criterion, which are, respectively, the variety of supported metrics, the time and memory needed, and the quality of the results, as follows.

Variety of supported metrics. The first criterion is concerned with the metrics the evaluation of a formalism yields, such as the resource utilization, service throughput, service latency, service jitter, queue sizes and resource utilizations. In principle, this depends on the selected underlying formalism, since the formalism has to accommodate the representation of these metrics in one way or another. For instance, a back-of-the-envelope model tends to support few met-

rics to ensure quick evaluation, whereas a TA is likely to provide support for more metrics. A PTA, which is an extension of a TA and thus more expressive, potentially provides support for even more metrics. Finally, an STA, which is an extension of both a PTA and a TA, provides support for the most metrics.

However, also the evaluation technique can be of importance, since some metrics, e.g., complete latency distributions (cf. Chapter 6 and 7), are not readily represented in the formalism and require extensive computations to be extracted from the model.

Time and memory. The second criterion refers to the amount of wall-clock time and computer memory needed to successfully execute the evaluation technique. Generally, they correlate positively, i.e., more wall-clock time goes hand in hand with more computer memory, and vice versa. However, exceptions do exist, e.g., performing long runs of discrete-event simulations takes much time, but does in general not require much computer memory. Namely, simulations tend to not store the whole model in memory opposed to, for instance, model checking.

Table 2.12: The applicability of evaluation techniques to underlying formalisms.

		Underlying formalism (Sect. 2.1)					
		LTS	**DTMC**	**TA**	**MDP**	**PTA**	**STA**
Evaluation tech. (Sect. 2.2)	**Sect.**	**2.1.1**	**2.1.2**	**2.1.3**	**2.1.4**	**2.1.5**	**2.1.6**
Back-of-the-envelope analysis	**2.2.1**	✓*	✓*	✓*	✓*	✓*	✓*
Discrete-event simulation	**2.2.2**	✓•	✓	✓•	✓•	✓•	✓•
Traditional model checking	**2.2.3**	✓	✓†	✓	✓†	✓†‡	-
Probabilistic model checking	**2.2.4**	✓	✓	✓	✓	✓‡	-

* = applicability may vary upon measure(s) of interest and model complexity.
• = resolving nondeterminism by the scheduler during evaluation.
† = replacing probabilistic by nondeterministic alternatives in the model.
‡= replacing stochastic residence times by alternatives with delays in the model.

Quality of results. The third criterion is concerned with the quality of results, which can be seen as the difference between obtained results and the true values. That is, evaluated results are results that have been generated using a certain evaluation technique and a formalism, whereas true values are values that would be obtained in reality, e.g., obtainable via measurements. Needless to say, they are exactly the same ideally. For the sake of simplicity, we will only consider latency values here and moreover assume that the modeled system displays deterministic latency values. Taking this into account, the quality of results is specified as a combination of three underlying properties, viz., accuracy, precision, and information, as follows.

The *accuracy* of results describes a systematical error. Namely, it is the degree of closeness of obtained results to a true value. The amount of accuracy can be defined as the absolute difference between the true value and evaluated value, where less difference means more precision. Additionally, Section 8.5.2 conveys the use of a relative difference (cf. Equation 8.9), which is scalable.

The *precision* of results describes random errors, which are a measure of statistical variability. That is, the degree to which repeated measurements yield the same obtained results. The amount of precision can be defined in terms of the difference between the minimum and maximum obtained values. In this case, a smaller difference indicates a higher precision. Regarding precision, we would like to point out that the results of discrete-event simulations (cf. Section 2.2.2) tend to vary, whereas iterations of model checking (Section 2.2.3 and 2.2.4) yield the same values for different executions. In chapter 7, we quantify precision (Equation 7.3 and 7.4).

The *information* of results is of concern when an evaluation technique returns ranges of possible latency values, e.g., an interval bounded by a lower and upper bound latency value. The amount of information ranges from no information, i.e., range $[0 : \infty]$, via some information, e.g., range $[3 : 5]$, to full information, e.g., the single value 6. The amount of information can be quantified as the difference between the lowest and highest value in the interval, where a smaller difference indicates a higher degree of information. The amount can also be quantified using the the relative difference (cf. Equation 8.9), similar to accuracy. Regarding information, we would like to point out that, for a model with nondeterminism, the results of discrete-event simulations (of Section 2.2.2) yield full information due to resolving nondeterminism in a random manner, whereas iterations of model checking (of Section 2.2.3 and 2.2.4) potentially yield no or some information. In chapter 7, we quantify information (Equation 7.1 and 7.2).

2.4 The plan of approach of this thesis

In this chapter, we have gathered background information to be used for our approach, which will be used for our approach in Chapter 4, 5, 6, 7, and 8. We use this information to make a number decisions about our toolset that implements the performance evaluation process (of Section 1.2), as follows.

Driven by earlier work on DSLs (of Section 2.5.1 and 2.5.2), we decide to use a DSL to represent the high-level performance model, because it leads to expressive and small models, which can be tailored to the domain of Medical Imaging Systems (MISs, [51, 164]) to be well understood by system designers in this domain. In line with previous work, we will use the DSL Framework Xtext [58] to create this DSL, which comes, among others, with a advanced Integrated Development Environment and the ability to export the DSL language and functionality to an Eclipse plug-in.

On the basis the literature study (of Section 1.3 and Appendix B), we have selected the Modest toolset (as described in Appendix B.6) to serve as the underlying model, because: (i) it supports and distinguishes between many transition systems (of Section 2.1); (ii) it provides multiple ways of analysis (as in Section 2.2); and, (iii) it has a generic language that is very expressive but also fairly abstract. We considered the alternatives Probabilistic Symbolic Model Checker (PRISM, [128, 165]) (of Appendix B.13) and UPPAAL (of Appendix B.15), but their languages turned out to be too low-level to model complex systems. Also, POOSL (of Appendix B.10) was a serious candidate for our approach because of its process algebra-based language and its extremely fast simulations [59]. However, POOSL does not provide model checking support (as appears in Section 2.5).

For evaluation, we will transform the concepts of the high-level performance model, as represented by the DSL language, into equivalents in the Modest language. This enables us to use different tools of the Modest toolset for various ways of analysis, while providing an understandable language to the system designer. Next, the results obtained via Modest are transformed into high-level results so that they match the level of the high-level performance model.

To communicate the results in a comprehensible way, we will use GNUplot [73,167] and GraphViz [61,77], which are widespread tools for generating visualizations. This yields a latency breakdown chart (see Figure 4.5), a latency bar graph (see Figure 4.6), and cumulative distribution functions of latencies (see Figure 4.7).

We intend to deliver a fully automated approach to minimize the efforts made by the system designer and reduce the number of mistakes made by the

system designer.

2.5 Gaining insight in Domain Specific Languages

In this section, we describe two case studies that have been performed in order to gain a better understanding of using DSLs for performance evaluation, namely a case study on collision prevention in Section 2.5.1, and a case study on IP in Section 2.5.2.

2.5.1 A Domain Specific Language for Collision prevention

This section is based on the following publication. The full version can be found in Appendix C.

[207] 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 *Lecture Notes in Computer Science*, pages 276–287. Springer, 2013. doi: 10.1007/978-3-642-40725-3_21

Prior to the actual work of this thesis, a case study on a Cyber-Physical System (CPS) for medical imaging has been conducted, resulting in a DSL for collision prevention [145], constructed using Eclipse [55] along with a DSL Framework Xtext [58]. This DSL has been constructed to prevent collisions between specific parts of the system, which is implemented by continuously performing distance queries on system parts and gradually slowing them down as soon as they are (too) close to each other.

Furthermore, the DSL comes with an automatic transformation to implementation code, which ensures that a CPS can easily be configured to behave as specified by a DSL instance without being bothered with low-level implementation code.

At the same time as the publication above, a transformation from the DSL [117] into Satisfiability Module Theorem (SMT) problems, to be solved by an SMT solver, has been constructed to perform validation checks, ranging from syntax checking to domain specific properties.

For performance, we constructed an automatic transformation from the DSL to a POOSL performance model [207], which can be evaluated using fast simulator Rotalumis to yield execution-time distributions. Using the transformation,

the performance of any CPS specified using a DSL instance can be evaluated. Besides the DSL instance, we found the performance to depend: on (i) the execution time of functions, which showed a great deal of variation; and, (ii) the specific use case performed, which affected which and how often functions were called.

To estimate execution times of function, we have performed benchmarks, i.e. measuring execution times via stopwatches that have been added to the implementation code, on a real CPS for distance queries on different system parts, and derived performance profiles, i.e., execution time distributions, from them. At the same time, we designed a number of use cases, i.e., set of object movements, to gain insight in which and how often functions execute. For instance, when the system parts do not move much, few distance queries get executed due to a caching mechanism that filters out similar queries.

We generated execution-time distributions for four use cases, using the fast simulator Rotalumis, and manually plotted these results using GNUplot. This turned out a labor-prone activity, especially when fundamental changes were made to the DSL instance.

To validate the generated performance model, we compared the execution-time distributions that have been generated via simulations with the measured performance profiles, for the four use cases. We automatically compared them using the widespread Kolmogorov distance and the execution ratio, which emphasizes how the execution times relate to each other relatively.

Furthermore, the semantics of the POOSL performance model were compared with semantics of the DSL in follow-up work [118]. They showed much resemblance, but also displayed differences that were the result of the interpretation of semantics.

Overall, we found the above procedure labor-intensive, although most of its individual steps were automated.

2.5.2 Insight in system design via modeling & simulation

This section is based on the following publication.

[90] S. Haveman, G. Bonnema, and F. van den Berg. Early insight in systems design through modeling and simulation. *Procedia Computer Science*, 28:171–178, 2014. doi: 10.1016/j.procs.2014.03.022

In a case study on IP for medical imaging, the creation of DSL was considered by building on previous work (of Section 2.5.1 and Appendix C).

However, it turned out that creating a transformation from this DSL to an underlying POOSL performance model is hard, due to the immense differences in complexities that needed to be bridged between the high-level DSL and the real complex IP system. To counteract this, we reduced complexity by extending the POOSL library with six new components, as follows.

1. *Generator:* a component that generates packages periodically with fixed inter-arrival times. This period is an instantiation parameter.
2. *Display:* a component that periodically displays an incoming package. Namely, the package that has been received last. The period is an instantiation parameter.
3. *Split-and-merge:* a component that merges n incoming packages into one package, and split this package in m outgoing packages, where n and m are instantiation parameters.
4. *Process:* a component that delegates packages to a processor, representing an operation on a package with an amount of work (load). The load is an instantiation parameter.
5. *Processor:* a component that processes packages at a certain speed (rate). The rate is an instantiation parameter.
6. *Mapping:* a connection between a process and a processor, in line with the Y-chart philosophy [13, 14].

It can be seen that these components are generic. Namely, packages are commonly used in computer science, but also applicable to the IP domain, as an image can be seen as a special kind of package.

Using the six components above, a transformation from a to be created DSL to the performance language was merely a matter of connecting different components and setting their instantiation parameters.

CHAPTER 3

Case studies on interventional X-ray systems

This chapter provides an overview of interventional X-ray (iXR, [158, 159]) systems as designed by our industrial partner Philips Healthcare [157]. Section 3.1 provides a system-level view of iXR systems, including their components, medical applications and settings. Section 3.2 provides insight in two important subsystems of iXR systems, viz., Movement Control and Image Processing (IP, [183]).

3.1 A system-level view on iXR systems

iXR systems enable minimally-invasive surgeries. While surgery is invasive by definition, many operations requiring incisions of some size are referred to as open surgery. Open surgery may leave large wounds that are painful and take a long time to heal. In contrast, minimally-invasive surgery refers to surgical techniques that limit the size of incisions needed and so lessens wound healing time, associated pain and risk of infection.

An iXR system consist of a number of parts (as depicted in Figure 3.1), as follows. An iXR system is used to assist a surgeon while performing surgery, during which a patient lies on a **table**. The iXR system displays a continuous stream of images of a patient on a **display**. These images are based on X-ray beams which are generated in the **arc** and caught by the **detector**, whose task it is to extract raw images from X-ray beams. Using the **control** panel, the surgeon can move the **arc** and **table** in various ways, e.g., tilt and rotate, and thereby change the angle of the recorded images of the patient, which are shown continuously on the **display**.

The parts of iXR systems come in many flavors and can be third party. For

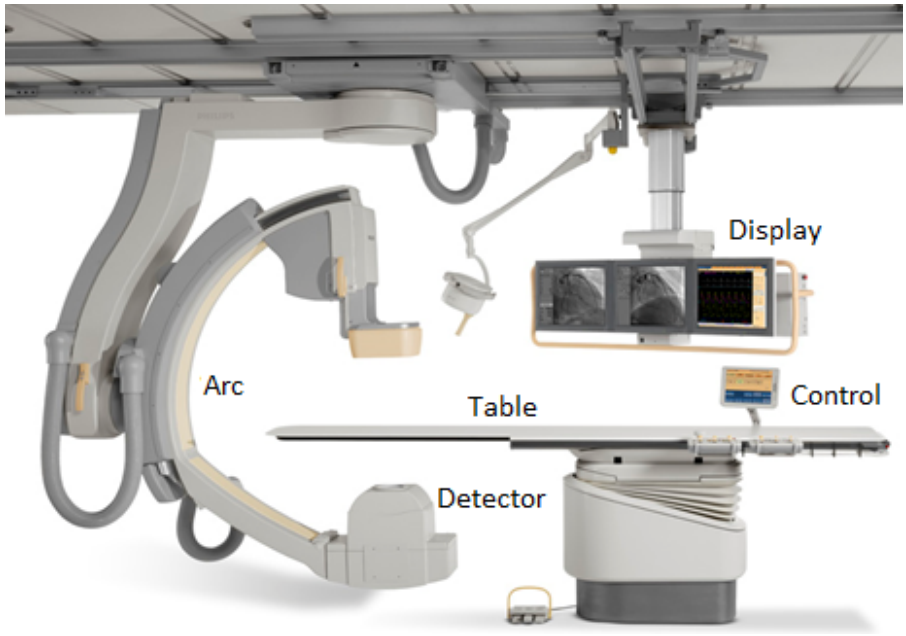


Figure 3.1: The main parts that constitute an iXR system, graphically depicted

instance, there are, among others, various combinations of tables, displays, and controls that constitute iXR systems. Under the hood, iXR systems are equipped with extensive software, such as an IP chain that combines various algorithms to enhance images, as well as software that takes care of collision detection and avoidance.

Next, we discuss some medical applications (in Section 3.1.1) and settings (in Section 3.1.2) of iXR systems to illustrate their high variability.

3.1.1 Different medical applications of iXR systems

iXR systems can be used for a wide array of medical applications [159], such as the so-called interventional cardiac, vascular, neuro, oncology, electrophysiology, and minimally invasive surgical procedures. We present the following three medical procedures to emphasize the variety of tasks iXR systems are used for.

Percutaneous Coronary Intervention [193] is a minimally invasive procedure used to treat the stenotic (narrowed) coronary arteries of the heart. During the procedure, a surgeon implants one or more stents in a patient. Stents are metal springs which are made to expand in the arteries to make them wider. For this purpose, an iXR system is used to retrieve high-quality 3D-images in real-time of the arteries, based on X-ray beams. Moreover, the iXR system generates a 3D-view of the arteries (see Figure 3.2(a)) that allows the surgeon to investigate the patient in great detail.

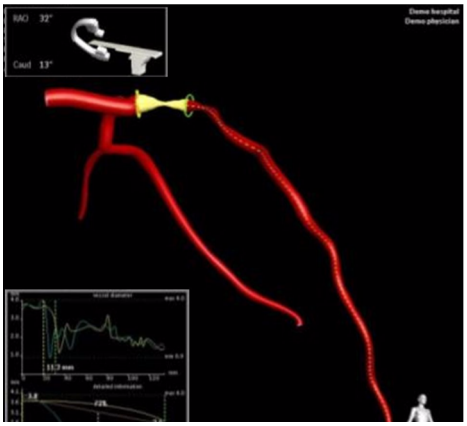
Percutaneous Aortic Valve Replacement [80] is the replacement of the aortic valve of the heart through the blood vessels. To prepare for this procedure, an iXR system is equipped with advanced software to create a map of the heart (as in Figure 3.2(b)). Consequently, the actual procedure, in which the aortic valve is implanted, is fast and involves minimal unwanted exposure to radiation.

Transesophageal echocardiography imaging [179] provides critical insights into soft tissue anatomy as well as function and flow information. An iXR systems fuses live echo and live X-Ray images in real-time (as is shown in Figure 3.2(c)). This leads to insight in the relationship between soft tissue anatomy and devices for fast and accurate interventions in structural heart diseases.

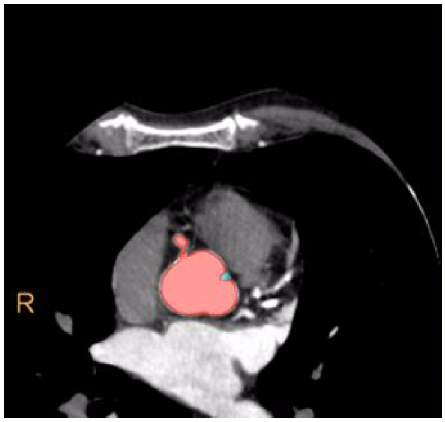
3.1.2 Different settings of iXR systems

To support different medical applications (e.g., of Section 3.1.1), iXR systems are expected to support, among others, the following settings that each customize an iXR system for a specific patient, surgeon and procedure.

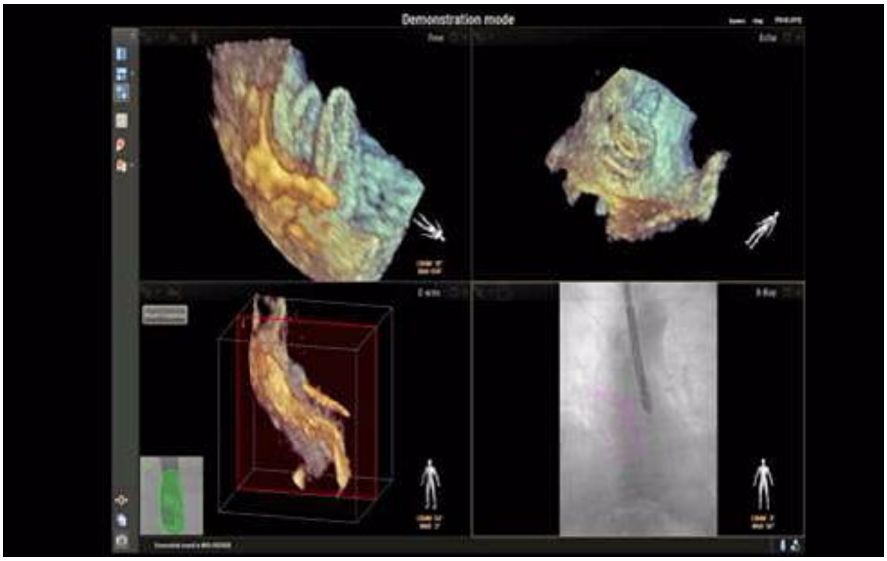
- Mono- or biplane: Using either one or two X-Ray beams to generate and detect images (see Figure 3.3). This corresponds to supporting 2D or 3D images, respectively.
- Image resolution: The number of pixels each generated image contains, e.g., square images of 512^2, 1024^2 or 2048^2 pixels.
- Image frame-rate: The constant frequency at which images are generated, e.g., 5, 10, 30, 50 or 100 images per second.
- Imaging algorithms: The way the commonly used IP steps are implemented, which affects the time to compute and the quality of the images dramatically.



(a) A 3D view of the arteries

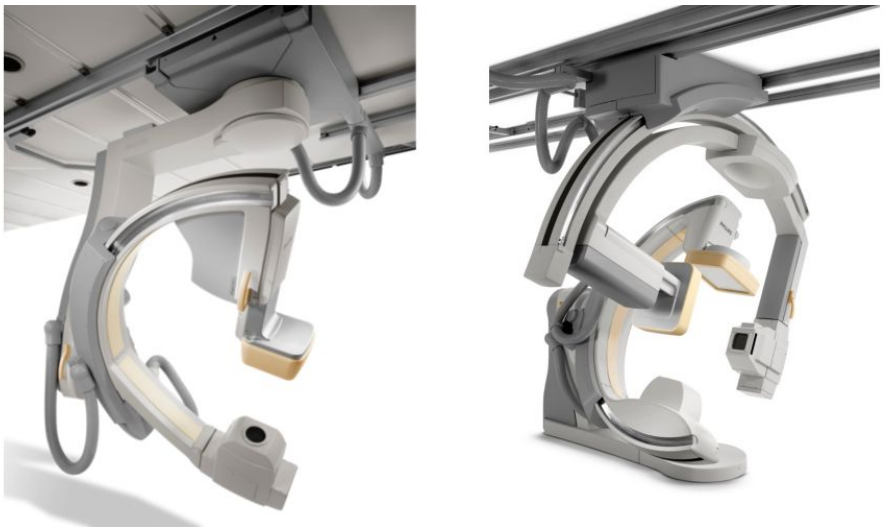


(b) An extensive map of the heart



(c) A Fusion of live echo and live X-Ray images

Figure 3.2: The output of iXR systems when used for three different applications



(a) A monoplane system with one detector (b) A biplane system with two detectors

Figure 3.3: A monoplane system compared to a biplane system

Note that selecting a medical application constrains which settings can be used, e.g., the procedure Percutaneous Coronary Intervention requires a biplane system (as in Figure 3.3(b)) to render 3D images and can therefore not be used with monoplane systems.

3.2 Two cardinal subsystems of iXR systems

Our case studies are based on two major components of iXR systems: (i) the Movement Control (MovCo) loop (explained in Section 3.2.1) monitors the location of the different physical parts of an iXR system and patient in physical space to make sure they do not collide; and, (ii) Image Processing (explained in Section 3.2.2) retrieves unprocessed images from the X-ray detector, which it then processes to enhance their quality, and finally sends to the monitor for display.

3.2.1 The Movement Control loop: collision prevention

Movement Control (MovCo) is concerned with collision prevention of the different moving parts of the iXR system (as depicted before in Figure 3.1) as well as the patient. For instance, a surgeon can move the **table** and **arc** parts of an iXR system via joysticks and buttons on the "control" panel to change the angle at which images are recorded (see also Section 3.2.2). This introduces a risk that these parts collide with each other and/or the patient. Besides the patient, whose safety should be guaranteed at all costs, also the **table** and **arc** parts are of prime concern. Namely, they have extraordinary sizes and weights, which makes it hard to slow them down and make them therefore prone to causing much physical damage to the system and its environment in case of a collision.

To enable collision prevention, the MovCo consist of a loop that repeats forever. In this loop, a number of distance queries are performed that each return the distance between the different parts of an iXR system. When two parts are critically close to each other in physical space, their freedom of movements is restricted by enacting speed limits upon them, which are distance dependent (i.e., brake patterns). This way the MovCo restricts the freedom of movement by a surgeon.

The MovCo can be seen as a trade-off between the following three aspects.

1 Functionality: The freedom of movements of the system parts, i.e., the reachable locations of parts, but also the speeds at which parts move.

2 Resources: The computational resources at hand to perform collision prevention computations, e.g., an off-the-shelf PC for distance queries.

3 Safety: The probability or possibility that two parts and/or the patient collide and the incurred damage of such a collision, e.g., a given algorithm guarantees that system parts never collide with the patient.

Given these three aspects, it is common to maximize functionality (allowing parts to move fast and in various positions), given a limited set of resources (depending on what the market price for an iXR system should be) and level of safety (prioritizing the safety of the patient).

Performance The safety of the MovCo is predominantly determined by performance, i.e., to detect a possible upcoming collision in time it is indispensable to frequently check the distances between the different parts of the iXR system and the patient. This means that the MovCo loop has to execute within a certain

time. Hence, it is essential to predict (the distribution of) the execution time of the MovCo loop to gain insight in the safety of the MovCo. The execution times of the MovCo loop vary a lot, because the underlying distance query algorithms have varying executions times [31]. They are, among others, caused by different object complexities and relative geometric positions (cf. Appendix C.3). To compensate for this, the MovCo is equipped with powerful hardware that runs at a relatively low utilization.

Case study A case study has been performed on the MovCo [207] to automatically generate execution-time distributions, based on an instance of an already existing DSL [145], performance profile, and use case (see also Section 2.5 and Appendix C).

3.2.2 Image Processing: hand-eye coordination

Image processing (IP) is responsible for turning X-ray based images into high quality images, in real-time. For this purpose, IP is represented in several parts of an iXR system (as depicted in Figure 3.1): IP retrieves unprocessed images from the X-ray **detector** and processes them to enhance their quality. Processed images are then send to the **display** to be observed by the surgeon.

IP comprises three subsequent categories of image operations:

1 Stripe processing: Operations on narrow image bands, e.g., detecting so-called dead pixels.

2 Image enhancement: Predominantly spatial and temporal noise reduction but also motion compensation.

3 Display processing: Preparing the image for the particular display at hand, e.g., image scaling

In line with literature [139], these steps are also referred to as pre-processing, processing and post-processing, respectively.

IP can be seen as a trade-off between:

1 Functionality: The constant quality and frame-rate of the images.

2 Performance: The throughput, latency and jitter of individual images.

3 X-Ray dose: The amount of X-Ray a patient and surgeon get exposed to, during a treatment.

4 Resources: The available computational and connectivity resources to process images. For instance, Central Processing Units (CPUs), Field Programmable Gate Array (FPGAs), and Graphical Process Units (GPUs), which are all connected via high-speed, dedicated networks.

Performance The safety of IP is mainly determined by performance, that is, to perform surgery on a patient, the surgeon needs to continuously receive high quality images that each have a small latency and jitter. The latency, the time between an image arriving on the detector and appearing on the display, needs to meet a requirement. Literature suggests an average latency below 165 milliseconds to enable hand-eye coordination [112], i.e., the surgeon perceives the images to be in real-time. The jitter, or variation in latencies, also needs to be below a certain value, which depends on the frame-rate, to ensure that all processed images are shown for about the same amount of time. Furthermore, the hardware of IP is expected to run at a reasonably high utilization due to the relatively constant execution times of image operations.

Case study A number of case studies haven been performed on image processing [90, 202–207], which have resulted into a language and toolbox to automatically assess the performance of iXR systems (see also Chapter 4, 5, 6 and 7 of this thesis).

CHAPTER 4

A Domain Specific Language for Performance Evaluation of Medical Imaging Systems

This chapter is based on the following publication.

[205] 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. doi: 10.4230/OASIcs.MCPS.2014.80

4.1 Introduction

Medical Imaging Systems (MISs, [51, 164]) are used to perform safety critical tasks. Their malfunctioning can lead to serious injury, and may even lead to fatalities [2]. The safety of an MIS is, among others, significantly determined by their performance, since imaging applications are time critical by nature. Predicting the performance of MISs 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, hence, can shorten the design cycle and time-to-market considerably.

Interventional X-ray (iXR, [158, 159]) systems are equipped with an Image Processing (IP, [183]) part (see Section 3.2.2), which can be seen as an MIS that dynamically records high quality images of a patient, based on X-ray beams.

These images need to be shown quickly for hand-eye coordination [112], viz., the surgeon perceives images to be real-time. 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. Due to their complexity, we are particularly interested so-called *biplane* iXR systems (see Figure 3.3(b)), which process two concurrent streams of images (named frontal IP and lateral IP) to enable 3D imaging. In this chapter, we investigate the use of a model-based approach to obtain insight in system performance.

For that purpose. we have decided to build *iDSL* [201], a Domain Specific Language (DSL, [142, 208]) and toolbox for performance evaluation of service systems. iDSL can be applied to MISs, because an MIS can be seen as a special kind of service system.

iDSL delegates performance analysis to Modest [83, 144] (cf. Section B.6) under the hood. Modest supports the modelling and analysis of Stochastic Timed Automata (STA, [32, 82]) by making use of Probabilistic Symbolic Model Checker (PRISM, [128, 165]) and UPPAAL [19, 132], as well as discrete-event simulation using MODES [83]. Modest has been selected because of the expressiveness of STA and its dual support for both model checking and simulation techniques. On the downside, the Modest language is still relatively low-level, which makes the modeling of iXR systems, which are complex. hard.

By design, iDSL adheres to the Y-chart philosophy [119], which separates the application from the underlying computing platform. It further uses hierarchical structures similar to the performance evaluation tool the Hierarchical evaluation tool (HIT, [18,199]), and can automatically generate design alternatives. Automated transformations from iDSL to different Modest model variants have been constructed, each taking full advantage of the capabilities of the underlying evaluation tools, e.g., UPPAAL and MODES. In a post-processing step, GraphViz [61, 77] and GNUplot [73, 167] are used to visualize performance outcomes.

This approach is not only very efficient, it also brings advanced formal performance evaluation techniques, e.g., based on model checking of timed automata and Markov chains, and discrete-event simulation, at the fingertips of system designers, without bothering them with the technical details of these.

Related work. [109,149] apply model checking with UPPAAL (cf. Section B.15) on real-time medical systems to address safety. Also, a study, in which PRISM is used, addresses quantitative verification of Implantable Cardiac Pacemakers

[34], which are time critical systems. [97,175] evaluate the performance of iXR systems using the Analytical Software Design (ASD) method.

The Octopus Toolset [13, 14] (cf. Section B.8) provides various tools for the modeling and analysis of software systems in general, whereas iDSL is specifically designed for service systems. [103, 104] address the performance evaluation of embedded systems. To the best of our knowledge they have not been used for evaluating the performance of service systems. In earlier work [90, 207] (as discussed in Section 2.5), simulation-based approaches using POOSL [59, 189] were proposed.

iDSL, as presented in this chapter, automates and connects the different stages of the performance evaluation process (see also Section A). This means that iDSL extends the related and earlier work, which generally puts emphasis on evaluating a performance model using one specific technique. Namely, iDSL automates the transformation of a high-level performance model into multiple underlying performance models, evaluates the performance of these models, and derives intuitive visualizations from the generated results.

Outline The remainder of this chapter is organized as follows. Section 4.2 introduces the high-level-performance model of iDSL. Section 4.3 shows the model transforms into the underlying performance model in Modest. Section 4.4 illustrates how iDSL models are evaluated. Section 4.5 presents the results that evaluation yields. Finally, Section 4.6 concludes the chapter.

4.2 The high-level performance model

This section describes the conceptual model of service systems, comprising six sections, that forms the basis of iDSL. Figure 4.1 depicts these six sections and their relations graphically.

A **service system**, or service-oriented system, as depicted in the upper right block, provides a service to one or more **service consumers** in its environment (exterior to the service system). In Figure 4.2, we show the possible interactions for a service system with two services and two consumers. Service systems deliver services in a flexible, dynamic and agile manner [150], like internet web servers, and IP systems. Service systems are commonplace in the domains of business, engineering and operations [121].

Within these systems, a **service** is an autonomous, platform-independent

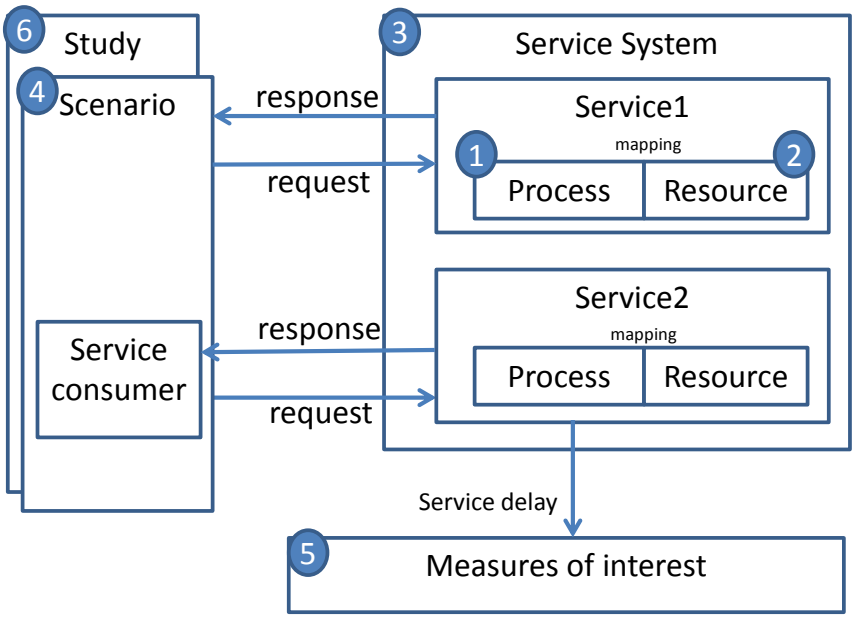


Figure 4.1: Conceptual model of a service system. External to the service system, measures of interest are obtained using scenarios.

entity that perform functions ranging from simple requests to computationally expensive processes, e.g., processing an image for quality enhancement.

A **service consumer** sends a request for a specific service at a given time, after which the system responds with some delay. Besides providing the proper functionality, i.e., returning the right answers to requests, service systems often need to meet performance constraints, e.g., the system has to reply to a request within a certain time, generally referred to as *latency*. Service systems can be hard to analyze when they handle many service requests in parallel, for multiple kinds of services, in a real-time manner.

To meet the constraints, a **service** is implemented using one or more processes, resources and a mapping, in accordance with the Y-chart philosophy [13, 14, 50, 119]. 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.

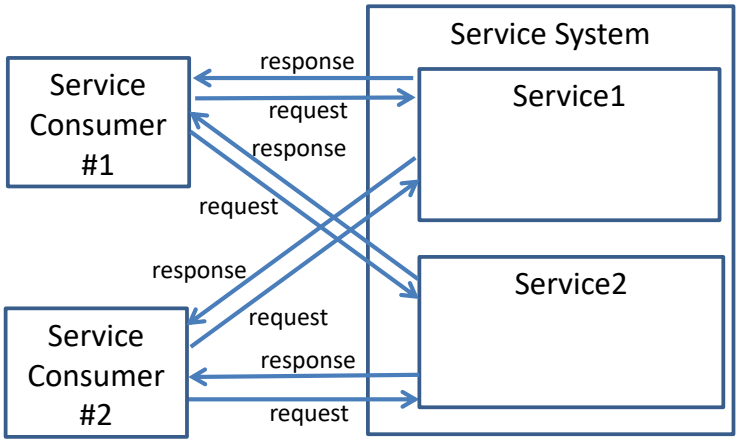


Figure 4.2: A service system with two services that interact with two consumers.

A **resource** is capable of performing one atomic task at a time, in a certain amount of time. For the sake of simplicity, resources that perform multiple tasks are not part of the model. When multiple services are invoked, the resources they make use of may overlap, causing concurrency and making performance analysis more challenging.

A **scenario** consists 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 due to concurrency.

A **study** evaluates a selection of systematically chosen scenarios to derive the system's underlying characteristics. Within a study, a **design space** is an efficient way to describe a large number of similar scenarios that only differ on a few aspects.

To conclude, a **measure of interest** defines what performance metric is interesting, given a system in a certain scenario. Measures can either be external to the system, e.g., service throughputs, latencies and jitters, but also internal, e.g., resource queue sizes and utilizations.

In the remainder of this section, we demonstrate how iDSL is used by implementing an example Image Processing (IP) system in it. For this purpose, we

use the six concepts of the iDSL language, as follows. Section 4.2.1 contains the process, Section 4.2.2 the resource, Section 4.2.3 the system, Section 4.2.4 the scenario, Section 4.2.5 the measure, and Section 4.2.6 the study.

4.2.1 The high-level Process

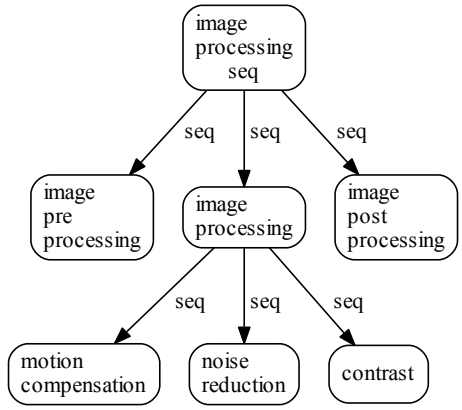
A process decomposes a service into a number of atomic tasks. This is implemented in iDSL using a recursive data structure with layers of sub-processes. At the lowest level of abstraction, the atomic tasks each specify a load, i.e., an amount of work, such as the required number of CPU cycles.

The process for the example (Table 4.1 and Figure 4.3(a)) combines hierarchies (curly brackets), sequential composition (*seq*) and atomic tasks (*atom*). At its highest level, it consists of a sequential task that decomposes into an atomic task “pre-processing" with (fixed) 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, during model execution, from a uniform distribution on [80,140]. The loads of these tasks are specified on the basis of analytics (as is quite common in performance evaluation [92]). In the following chapters (Chapter 5, 6 and 7), we will show that loads can also be based on measurements that have been performed on real iXR systems.

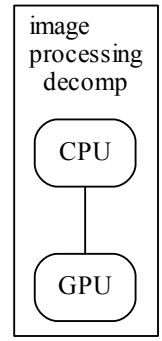
Additionally, iDSL supports the process algebraic constructs for parallelism (*par*), nondeterministic choice (*alt*) and probabilistic choice (*palt*), as well as a mutual exclusion (*mutex*) to permit at most one process instance at a time to enter a certain process part.

Table 4.1: The code of an iDSL process

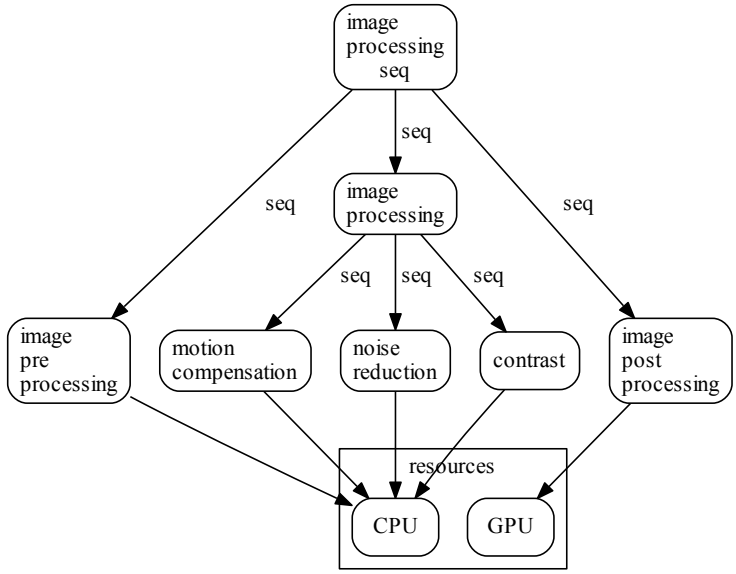
```
Section Process
ProcessModel image_processing_application
 seq image_processing_seq {
  atom image_pre_processing load 50
  seq image_processing {
   atom motion_compensation load 44
   atom noise_reduction load uniform(80:140)
   atom contrast load 134 }
  atom image_post_processing load 25 }
```



(a) IP ProcessModel



(b) IP ResourceModel



(c) IP service

Figure 4.3: Visualizations for process, resource and service, automatically generated from iDSL code.

Table 4.2: The code of an iDSL resource

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

4.2.2 The high-level Resource

In iDSL, a resource is defined as recursive 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 that have a rate that specifies how much load they can process per time unit, e.g., the number of CPU cycles per second. Resources need to be connected to perform operations in sequence for one process, which is enforced via high-level input validations. The connections further enhance the way resources are visualized.

We model the resource in our example as a composite resource (Table 4.2 and Figure 4.3(b)). It consists of two atomic resources, i.e., a "CPU" with rate 2 and a "GPU" with rate 5, that are connected to each other. A connection is in the iDSL code is used to visualize the connection of resources, e.g., CPU and GPU Figure 4.3(b), and for input validation, i.e., atomic tasks that execute in sequence need to be mapped to resources that are connected.

Table 4.3: The code of an iDSL system

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

Table 4.4: The code of an iDSL scenario

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

4.2.3 The high-level System

A system provides one or more services to its environment. In our example (Table 4.3), we construct an overall system with one service. In line with the Y-chart philosophy [13,14], a service (Figure 4.3(c)) connects the already defined process (Figure 4.3(a)) and resource (Figure 4.3(b)), via a static mapping that assigns atomic tasks of the process to atomic resources (the unlabeled arrows). This decomposition makes it possible to conveniently change the process and/or resource part of the service, e.g., in order to conveniently employ Design Space Exploration (DSE).

4.2.4 The high-level Scenario

A scenario is defined by a number of bundle of service requests to a system, each individually requested over time (Table 4.4). A *biplane* iXR system comes with two kind of services, viz., frontal IP and lateral IP, which are defined, as follows.

For frontal IP, 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.

For lateral IP, the times of the request are defined similarly, i.e., inter-request times are a constant 400. However, to influence the degree of concurrency, we add an initial offset by using two *dspace* function calls that are constant within a design instance. This means that an iDSL model specifies multiple design instances, as will be explained further in Section 4.2.6.

4.2.5 The high-level 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 tech-

niques to obtain them. To illustrate our approach, we specified two measures (in Table 4.5), as follows.

First, measure "ServiceResponse times" is used to obtain service response times for a given number of runs of a certain length (1 run of length 280 in the example). The measure also provides insight in resource utilizations and latency breakdowns. Under the hood, they are obtained, in one go, via **simulations** (of Section 2.2.2).

Table 4.5: The code of an iDSL measure

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

Second, measure "ServiceResponse absolute times" leads to the absolute minimum and maximum response times, given a system and scenario. It does not require parameters in the iDSL language. Under the hood, they are obtained via **model checking** (of Section 2.2.3).

4.2.6 The high-level Study

Finally, a study summarizes a collection of scenarios that one would like to analyze in an automated manner. In our example (Table 4.6) we use one scenario as a basis that is parametrized using a so-called design space.

A design space is 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 with variable offset and enumerate the desired values. After this, the *dspace* function can be used for the offset parameter as done twice in the scenario section (Table 4.4).

4.3 The underlying performance model

In Section 4.2, the use of iDSL was demonstrated by implementing an example Image Processing (IP) system in it, using the six core concepts of the iDSL language. In order to provide semantics to the iDSL language, iDSL automatically transforms each concept into corresponding Modest [83] code. In this section,

Table 4.6: The code of an iDSL study

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

we provide the generated Modest code of the example IP system for each of these six concepts, as follows. Section 4.3.1 covers the process, Section 4.3.2 the resource, Section 4.3.3 the system, Section 4.3.4 the scenario, Section 4.3.5 the measure, and Section 4.3.6 the study.

4.3.1 The underlying Process

Section 4.2.1 provides the iDSL process of an example IP system (see Table 4.1) in which hierarchies, sequential composition and atomic tasks are combined.

iDSL transforms it into Modest code (see Table 4.7) using the following guidelines. First, hierarchies in iDSL are implemented using layered processes without parameters, e.g., as with process *image_processing_seq()*. Second, sequential composition in iDSL is converted to its equivalent in Modest, i.e., using semicolons. Third, atomic tasks become references to single functions in Modest (which are defined in Section 4.3.3) that represent the mapping, with their respective loads as parameter. Finally, each process waits for a generator to be triggered through binary communications (to be explained in Section 4.3.4), indicated by a question mark.

4.3.2 The underlying Resource

Section 4.2.2 describes the iDSL resource of an example IP system (in Table 4.2) in which two connected resources are defined, viz., the CPU and GPU.

iDSL transforms it into Modest code (see Table 4.8) by creating two Modest processes per iDSL resource, as follows. The first process (with prefix "machine_") comprises binary communications to handle concurrency and a delay that represents the resource being in use, i.e., processing a process request. The self-recursion ensures that the resource stays alive forever. The delay represents task time, which is the quotient of the load and rate, e.g., the quotient of CPU cycles and CPU cycles per second is given in seconds. The second process (with prefix "machine_call") abstracts communications (indicated with ? and !) from

Table 4.7: The generated Modest code from an iDSL *process*

```
process image_processing_application_instance(){
  generator_image_processing_application?;
  image_processing_seq()
}

process image_processing_seq(){
  image_pre_processing(50);
  image_processing();
  image_post_processing(25)
}

process image_processing(){
  motion_compensation(44);
  noise_reduction(Uniform(80,140));
  contrast(134)
}
```

Table 4.8: The generated Modest code from an iDSL *resource*

```
process machine_CPU(){
  real taskload;
  machine_CPU_start? {= taskload=sync_buffer =};
  delay (taskload / 2 ) machine_CPU_stop!;
  machine_CPU()
}

process machine_call_CPU(real taskload){
  machine_CPU_start! {= sync_buffer=taskload =};
  machine_CPU_stop?
}

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

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

Table 4.9: The generated Modest code from an iDSL *system*

```
process image_pre_processing(real taskload){
  machine_call_CPU(taskload) }

process motion_compensation(real taskload){
  machine_call_CPU(taskload) }

process noise_reduction(real taskload){
  machine_call_CPU(taskload) }

process contrast(real taskload){
  machine_call_CPU(taskload) }

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

the process layer. In the given Modest code, concurrency for resources is represented using nondeterministic choices, in a non-preemptive manner. On one hand, nondeterministic scheduling is easy to implement and yields a smaller state space, but on the other hand, it does not per se take properties like starvation and fairness into account. In the next chapter (cf. Section and 5.2.3 and 5.3.3), we equip a resource with buffers to enable various way of scheduling.

4.3.3 The underlying System

Section 4.2.3 describes the iDSL system of an example IP system (in Table 4.3) in which one only service is defined. iDSL transforms this system description into corresponding Modest code (see Table 4.9) by creating processes (which are referred to in Section 4.3.1) for each mapping. Each of these processes calls the mapped resource in Modest.

4.3.4 The underlying Scenario

Section 4.2.4 describes the iDSL scenario of an example IP system (in Table 4.4) in which the service is called via two infinite streams of requests, for respectively frontal IP and lateral IP.

In Modest (in Table 4.10), we show the situation for offset=200. The Modest code for other designs is very similar. Namely, one only has to replace the three boldfaced occurrences of 200 by the offset value of the respective design.

The Modest code comprises four Modest processes, viz., two processes for each stream of requests. The first process (with prefix “init_generator”) performs the initial delay once, e.g., 0 and 200, respectively, in the example. The second process (with prefix “generator) then loops forever, with period of the inter-request time, e.g., 400 for both in the example, triggering the process once every loop. Additionally, when the service system fails to respond to a request immediately, a time-out occurs that drops the request.

4.3.5 The underlying Measure

Section 4.2.5 describes the iDSL measure of an example IP system (in Table 4.5) in which two measures are defined. They are evaluated as follows.

First, measure “ServiceResponse times” is used to obtain service response times for a given number of runs of a certain length (1 run of length 280 in the

example). Additionally, the measure also provides insight in resource utilizations and latency breakdowns, i.e., the latencies of subprocesses, without any extra efforts. The measure leads to the generation of a Modest model with STA as its underlying model and it is evaluated via discrete-event simulations (of Section 2.2.2, using MODES [83] of the Modest toolset.

The service latencies and resource utilizations are retrieved by extending

Table 4.10: The generated Modest code from an iDSL *scenario*

```
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()
}

process init_generator_image_processing_service2() {
  delay
    (200) generator_image_processing_service2()
}

process generator_image_processing_service2() {
  clock c; tau = c=0 =;
  alt
  :: generator_image_processing_application!
  :: delay(1) tau // time-out ;
  when
    urgent(c >= ((400 + 200) - 200) )
      generator_image_processing_service2()
}
```

the already given code with both measurement points and properties (as illustrated in Table 4.11). Each (sub)process is enclosed by a stopwatch (with prefix "_stopwatch") to register a latency value, which are each retrieved via a Modest property. Additionally, resource utilizations are obtained via a property that returns a utilization after a carefully selected time, e.g., after 10000 time units (boldfaced) in the example. For this purpose, a resource has been augmented with a variable (with prefix "_util_counter") that keeps track the cumulative processing time.

Second, measure "ServiceResponse absolute times" 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 measure leads to the generation of a Modest model that adheres to Timed Automata (TA, see Section 2.1.3) as its underlying model by (among others): (i) turning real numbers into integers by rounding, and (ii) turning probabilistic alternatives into nondeterminstic alternatives by omitting probabilities. The model is evaluated via model checking (of Section 2.2.3) using Model Checking Timed Automata (MCTAU, [24, 83]) of the Modest toolset, which in turn uses UPPAAL [19, 132] under the hood.

The measure triggers iDSL to run a binary search algorithm that is used to obtain the absolute minimum and maximum latency (see Section 4.4.4 for more detail), using multiple executions of MCTAU. Each process has stopwatches (as with simulation), but the image counter is omitted to make the state space finite. However, as a consequence it is not possible to determine which service requests, e.g., the third service request, yield the minimum and/or maximum latencies.

Note that iDSL generates different Modest code for each measure, to take full advantage of the STA expressiveness and the TAs model checking capability.

4.3.6 The underlying Study

Section 4.2.6 shows the iDSL study of an example IP system (in Table 4.6). It comprises a design space with parameter "offset" with seven possible values.

Modest code (see Table 4.12) is generated for each design that initiates the generated Modest processes introduced so far, in parallel, as follows:

1. Process instances: two Image Processing instances (cf. Section 4.7),
2. Services: two Image Processing services (cf. Section 4.4),
3. Resources: resource CPU and GPU (cf. Section 4.8).

Table 4.11: The generated Modest code from an iDSL *measure*

```
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 );

property property_utilization_CPU =
  Xmax ( util_counter_CPU / 10000 | time == 10000 );

property property_utilization_GPU =
  Xmax ( util_counter_GPU / 10000 | time == 10000 );

process machine_CPU() {
  ...
  delay
    ( taskload/ 2 )
      tau {= util_counter_CPU += ( taskload / 2 ) =};
  ...
}

process machine_GPU() {
  ...
  delay
    ( taskload/ 5 )
      tau {= util_counter_GPU += ( taskload / 5 ) =};
  ...
}
```

Table 4.12: The generated Modest code from an iDSL *study*

```
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()
}
```

The Modest code (of Table 4.12) is the same for all seven designs, since it does not contain design specific elements.

4.4 Performance model evaluation

This section covers the functioning of our iDSL tool (as illustrated in Figure 4.4). iDSL requires two user roles to be fulfilled, viz., the *modeller* and the *analyser*. The modeller constructs a model of a real system and the analyser specifies measures to perform (of Section 4.4.1). Execution of the model (of Section 4.4.2) then yields artefacts with performance results (of Section 4.5), to be investigated by the analyser.

4.4.1 Modelling

A modeller and analyser interactively create an iDSL instance in the Eclipse IDE for DSL Developers [56, 78], adhering to iDSL's grammar. Input validation comprises syntax checking and more 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. This way of modelling leads to a syntactically valid iDSL model.

4.4.2 Execution

The execution of an iDSL model takes place via a so-called tool chain (as depicted in Figure 4.4), as follows. First, Modest models are executed in MODES (cf. Section 4.4.3) yielding, among others, latencies and utilizations. Second, some other Modest models are analyzed using UPPAAL (cf. Section 4.4.4), via the model checker MCTAU of the Modest toolset [24], yielding *absolute latency bounds*. Finally, visualizations are generated (cf. Section 4.4.5), primarily using Graphviz [61, 77].

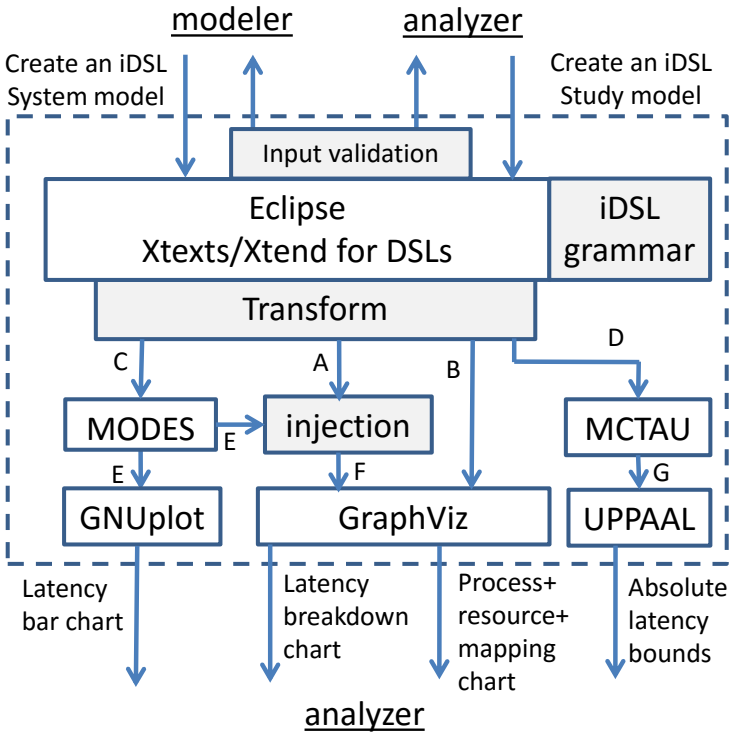


Figure 4.4: The iDSL tool chain overview. A modeller and analyser create an iDSL model. The iDSL tool transforms this model into Modest and GraphViz models, leading to performance measures to be evaluated by the analyser.

4.4.3 Execution of simulation

In this subsection, we describe how simulation (part of the execution chain of Section 4.4.2) is implemented in iDSL.

First, an iDSL model is automatically transformed into one or more Modest models (cf. Figure 4.4, C). These transformations are written in the Xtend programming language [57] and generate text output, based on iDSL instance constructs.

In turn, these Modest models are executed in the MODES simulator and yield latencies and utilizations (cf. Figure 4.4, E), which serve two purposes. First, the high-level latencies per instance are transformed into a *latency bar chart* using GNUplot. Second, the low-level latencies and utilizations are used to create a so-called latency breakdown chart (more carefully explained in Section 4.4.5).

To eliminate nondeterminism, an “as soon as possible” (ASAP) scheduler for time is used, and a uniform resolution for nondeterministic choice [87], which are fixed parameters that iDSL explicitly provides to the simulator 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.

4.4.4 Execution of model checking

In this subsection, we describe how model checking (part of the execution chain of Section 4.4.2) is implemented in iDSL.

First, an iDSL model is automatically transformed into one or more Modest models (cf. Figure 4.4, D). These Modest models are executed using the model checker MCTAU [24] of the Modest toolset, which in turn delegates analysis to UPPAAL [19, 132] under the hood.

The Modest models are simplified to compensate for the limitations model checking brings about, i.e., model checking as provided by UPPAAL does not allow the use of real numbers, probabilistic choices, and many variables that cause the state space to explode. Hence, iDSL automatically "downgrades" STA to TAs. For this purpose, real numbers are replaced by integer approximations. To minimize the effect of rounding as much as possible, it is recommended to use small units in the model. Additionally, probabilistic choice and infinite distributions are replaced by nondeterministic choice. Finally, some performance measuring variables (cf. Section 4.11) are removed to limit the state-space size. For instance, the uniform function in the process (see Table 4.7), represented

Table 4.13: Binary search for bounds, pseudo code

LB: Compute lower bounds, pseudo code

```
LB (lbound,ubound){
  if (abs(ubound-lbound)<=1)
    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)
}
```

LB: 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) -> 159
```

by a continuous probability function in STA, becomes a nondeterministic, finite (integer-valued) choice in TAs.

As TAs only support properties with boolean expressions, the absolute values cannot be retrieved using single properties. Therefore, iDSL comes with a binary search algorithm that leads to a solution with time complexity $O(log(n))$, where n is the size of the search range. The algorithm consists of two functions, i.e., an LB function to compute lower bound values and an UB function for upper bound values. The lower and upper bounds are *valid* when they are, respectively, lower and higher than all possible outcomes and *strict* when also the distance between them is minimal, i.e., the lower bound is the highest valid one and vice versa. iDSL returns bounds that are both valid and strict.

LB is a recursive function (Table 4.13, top) with two parameters, the lower and upper bound of the current range of values, which are both integers. The stop criterion, i.e., the lower and upper bound value are close enough to each other (their difference is smaller than or equal to one) 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 either the lower or higher 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 has complexity $O(\log(n))$, the choice of n does therefore not affect the workload too much, i.e., it is advised to overestimate the size of the range in case of doubt. However, it is not always straightforward to find a reasonably tight upper bound (see Section 7.5.4, E). Based on simulation results, we choose $[0 : 1024]$ to be our initial range, where value 1024 is assumed to be larger than all conceivable latencies. The execution trace (Table 4.13, bottom) conveys 12 recursive calls before the final value 159 is obtained. This means that the image processing system will never display a service response time smaller than 159, in the given scenario.

4.4.5 Execution of visualization

An iDSL model is automatically transformed into two kinds of GraphViz specifications (cf. Figure 4.4, A+B). These transformations are also written in Xtend and generate text output, based on iDSL instance constructs.

Some GraphViz specifications (in particular the ones created via Figure 4.4, B) are performance unrelated and provide a visual presentation of the processes, resources and mappings of the system (as already shown in Figure 4.3). They are turned into a *process+resource+mapping chart* using the GraphViz tool.

The remaining GraphViz specifications (resulting from Figure 4.4, A) have designated placeholders to contain performance numbers and form the first input of the *injection* step. The latencies at different process levels and resource utilizations (cf. Figure 4.4, E) form the second input of the *injection* step. In the injection step, both inputs are combined; iDSL replaces the placeholders by their respective performance numbers, using the Find And Replace Text tool [140]. This leads to a new GraphViz specification, which is forwarded to the GraphViz tool (cf. Figure 4.4, F) and transformed into a *latency breakdown chart* (see Figure 4.5). In the next chapter, we extend the latency breakdown chart with coloring for enhanced understandability (as can be seen in Figure 5.6).

To illustrate the meaning of this 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 uti-

lization 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. Two services are each invoked periodically every 400 time units. Therefore, the “GPU" has a utilization of $(5+5)/400 = 0.025$.

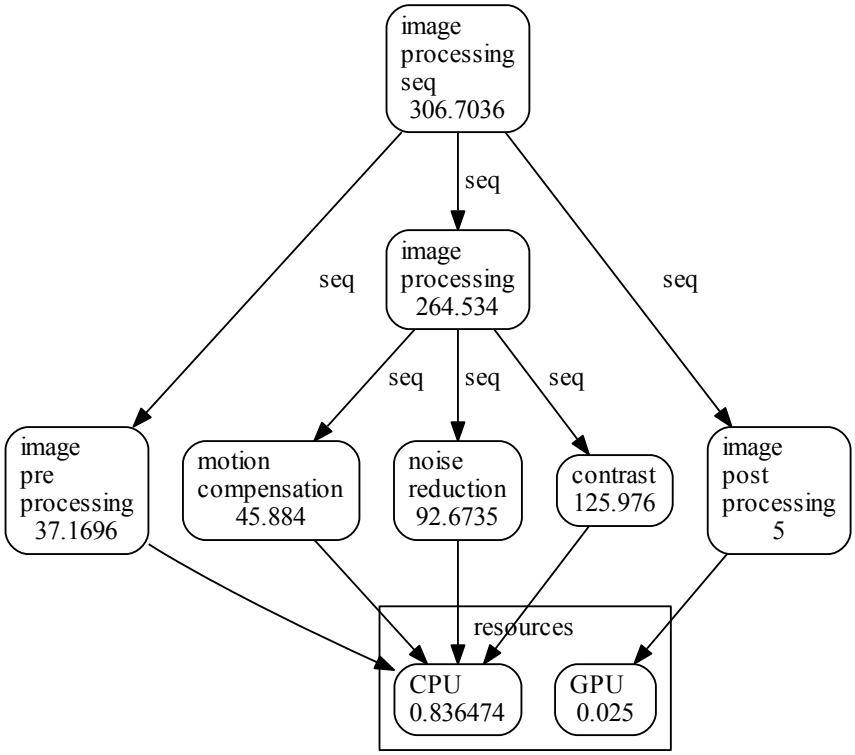


Figure 4.5: The latency breakdown chart and utilizations (offset=0), based on MODES simulation results. It is automatically, including the execution of required simulation runs, generated from the iDSL code.

4.5 Performance results

Using the presented tool chain, iDSL offers the possibility to compare several design alternatives from various perspectives, in an automated manner. This does not only save the system design a lot of work, but also allows for the comparison of many designs. We proceed with discussing the results iDSL can generate. First, we discuss the results based on MODES simulations in Section 4.5.1, after which we review results obtained from model checking using MCTAU and UPPAAL in Section 4.5.2.

4.5.1 Simulation results

We have defined a study with seven design alternatives. iDSL automatically generates a latency breakdown chart and a latency bar graph for each one of them. We present the ones for the offset=0 case (cf. Figure 4.5 and 4.6). As can be seen in Figure 4.6, which shows the observed latencies per image, the latency varies highly. This is due to a high degree of concurrency, viz., offset=0 implies that both services execute exactly at the same time, which forces the scheduler to make many concurrency resolving decisions that increase the variability.

We further see in the latency breakdown chart of the next chapter (cf. Figure 5.6) 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 4.7). As expected, the latency times are high when the offset is small and the level of concurrency large, and vice versa. Consequently, no concurrency takes place for the highest offsets (viz., offset=120, offset=160, and offset=200), because both imaging chains are far enough apart to avoid concurrency.

4.5.2 Model checking results

We applied MCTAU on a system with one image processing service to reduce complexity, which implies no concurrency. Therefore, the results are similar to the simulation cases without concurrency, viz., offset=120, offset=160, and offset=200. The computation of the lower (Table 4.13, bottom) and upper bound leads to values 159 and 189, respectively. As required by definition, all simulation outcomes fall within the absolute bounds as computed by model checking.

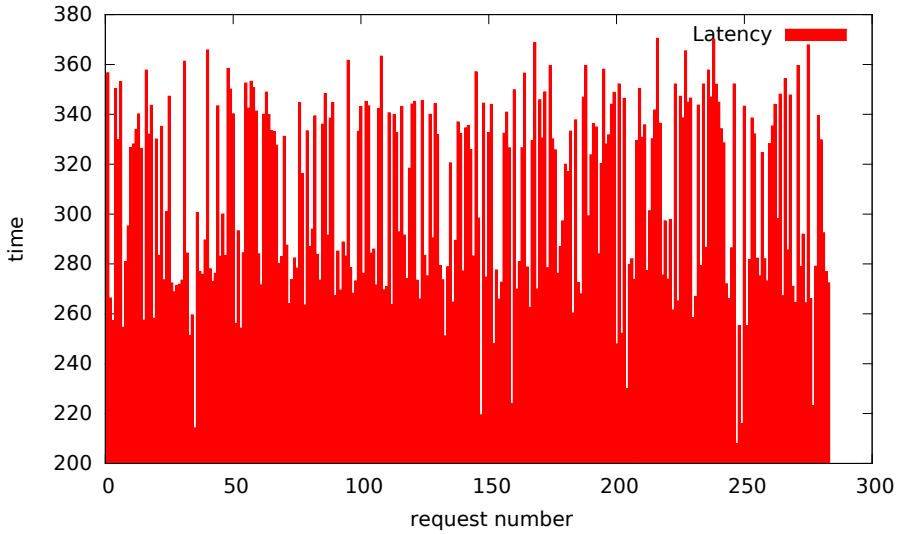


Figure 4.6: The MODES latency times bar graph (offset=0) for 280 service requests, which is automatically generated from the iDSL code.

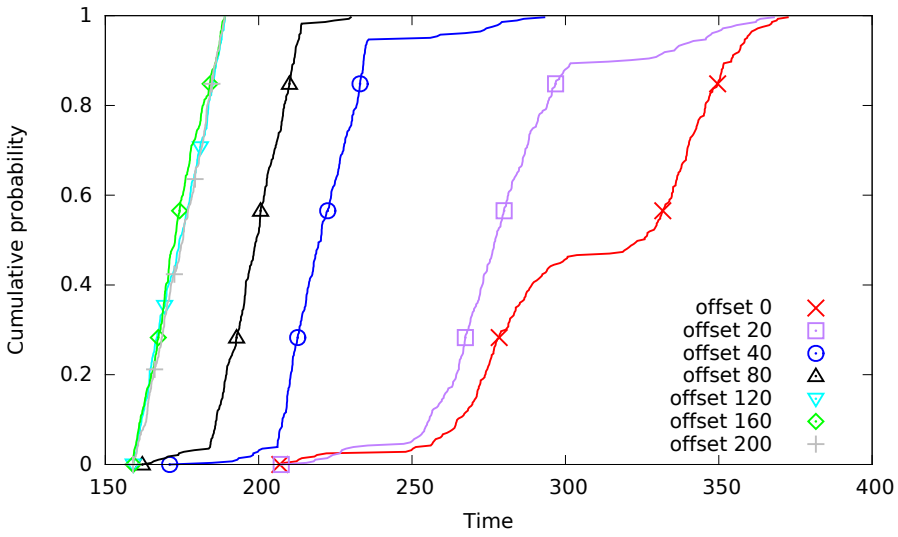


Figure 4.7: The cumulative distribution functions of latencies for seven design instances, automatically generated from the iDSL code.

4.5.3 Validation via back-of-the-envelope computations

We perform back-of-the-envelope computations to gain insight in the validity of the just presented results. We consider the case of no concurrency and concurrency, as follows.

In case of no concurrency, the execution time of a service can be computed as the quotient of each individual load in the process (of Table 4.1) and the rate of the resource (of Table 4.2) it is mapped to (of Table 4.3). This yields

$$\frac{50 + 44 + min(80, 140) + 134}{2} + \frac{25}{5} = 159,$$

and

$$\frac{50 + 44 + max(80, 140) + 134}{2} + \frac{25}{5} = 189,$$

for the minimum and maximum time, respectively. The difference is caused by the uniform distribution of "noise_reduction", which is represented using a *min* and *max* function in the above computations. To conclude, we observe that the result is in line with the absolute bounds (of Figure 4.8).

In case of concurrency, two services with potentially many service instances are being executed in the system. However, since the period between the instantiation of service instances of an individual service is 400 time units, and services maximally execute for 189 time units, we may conclude that maximally two service instances are executing at a time. Consequently, we multiply the results for the no concurrency case by two to obtain an overestimation of the upper bounds. This yields $159 \cdot 2 = 318$ and $189 \cdot 2 = 378$ for the minimum and maximum time, whose difference again depends on the uniform distribution of "noise_reduction". Figure 4.7 shows indeed that these outcomes are a valid and fairly strict upper bound for each of the design instances.

In short, validation has shown that four valid but also fairly strict bounds can be obtained via a back-of-the-envelope computation. This does not prove the validity of the computed results, but does increase confidence in them.

4.6 Conclusion

In this chapter we have presented iDSL, a DSL and toolbox for the performance evaluation of service systems. We applied iDSL to a small example on MISs, which are a special kind of service systems, to assess the feasibility of iDSL.

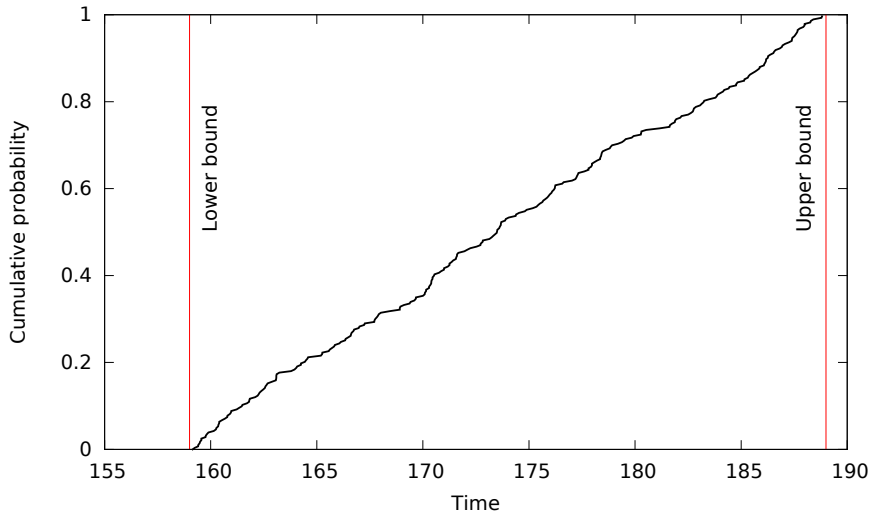


Figure 4.8: The absolute minimum and maximum bounds (of a system with only one image processing service) obtained via model checking and a CDF of the simulation outcomes (for the same system), which are automatically generated from the iDSL code.

iDSL automates performance analysis by delegating smaller performance evaluations to Modest [83] (cf. Section B.6) under the hood. iDSL uses thereby simulations to obtain a variety of results such as service latencies and resource utilizations, and model checking for obtaining absolute latency bounds. iDSL is capable of returning these results for a number of designs. iDSL also displays its results visually, such as latency breakdown charts and latency bar charts, by delegating results to the commonly known tools GraphViz and GNUplot. On top of this, iDSL accomplishes all of the above, i.e., a trajectory from iDSL models to all of the above artifacts, in a fully automated manner.

To increase the applicability and the quality of generated results of iDSL, we apply iDSL to more extensive use cases in the following chapters. In Chapter 5, we extend the iDSL process and tool chain to enable model calibration based on real measurements. In Chapter 6, we add a measure to iDSL to obtain latency distributions via probabilistic model checking, which is made more automated and efficient in Chapter 7. Finally, we apply iDSL to a different domain than iXR system, viz., energy-aware load balancers, in Chapter 8.

CHAPTER 5

Automated Performance Prediction and Analysis of Medical Imaging Systems

This chapter is based on the following publication.

[206] 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. doi: 10.1007/978-3-319-23267-6_15

5.1 Introduction

In this chapter, we again investigate the performance of *biplane* interventional X-ray (iXR, [158, 159]) systems (see Figure 3.3(b)), which are used to perform safety critical tasks. Biplane iXR systems comprise two imaging chains that are positioned in perpendicular planes to enable 3D-imaging and are currently implemented using two separate hardware platforms. However, for various reasons, e.g., costs, physical space, energy consumption and failure rate, it is worth investigating running the software for both image chains on shared (but more powerful) hardware. To assess its feasibility, we predict the performance for shared hardware and compare it to the former case of separate hardware. We would like to point out that sharing hardware gives potential to contention, which may result in increased latency and jitter of images, which, in their turn, affect (perceived) system safety.

In Chapter 4, we introduced the iDSL language and toolchain for evaluating the performance of service-oriented systems. The iDSL language instances em-

bodies a performance model of service systems, while the iDSL tool chain transforms iDSL instances into Modest language equivalents that are evaluated using the Modest toolset for multiple design instances. This yields performance numbers and intuitive graphs, e.g., a latency breakdown charts. However, iDSL has only been tested on a small example of iXR systems in Chapter 4. Namely, the process loads were specified using so-called "guestimates", mixtures of guesswork and calculations. Consequently, it is still unknown whether iDSL can be used in a more realistic, complex setting.

In this chapter, we therefore extend the performance evaluation approach of Chapter 4 to automatically go from real measurements, via a formal model, to performance predictions, for many different designs. Starting point are models expressed in the iDSL language, which we extend with model calibration support, i.e., the system designer can inject measurements (e.g., performed on an earlier version of the system) into the model to make it reflect reality more. Next, iDSL models are automatically transformed into Modest language [88] equivalents to be evaluated using the Modest toolset. Additionally, iDSL generates visualizations using Graphviz [61, 77] and Gnuplot [73, 167], including newly supported trade-off graphs.

In Section 1.3, a number of performance evaluation indicators were introduced that are elaborated in Appendix A. For this chapter, we have identified five objectives, inspired by indicators $\mathcal{I}1.3$, $\mathcal{I}3.2$, $\mathcal{I}3.6$, $\mathcal{I}4.2$, and $\mathcal{I}4.3$, respectively, that a performance evaluation approach should meet, i.e., it should

$\mathcal{I}1.\mathbf{3}$ use as few costly measurements as possible;

$\mathcal{I}3.\mathbf{2}$ be able to automatically evaluate a large number of complex designs;

$\mathcal{I}3.\mathbf{6}$ be applicable to real complex systems;

$\mathcal{I}4.\mathbf{2}$ present its results intuitively via understandable, aggregated metrics; and,

$\mathcal{I}4.\mathbf{3}$ present its results visually.

In this chapter, these objectives are realized through the following five respective contributions. First, we extend the iDSL process with functionality to calibrate the model based on measurements. That is, Empirical Cumulative Distribution Functions (ECDFs) are generated, which are discrete distributions that are generated on the basis of a finite set of measurements. We also add support to iDSL for deriving measurement predictions from these measurements, which minimizes the need for costly measurements. This is in contrast to mainstream Design Space Exploration (DSE) approaches that tend to rely on many measurements [13, 14, 104].

Second, we reuse the simulation-based measure of the previous chapter (see Section 4.4.3). We determine whether the measure scales well to quickly evaluate individual design instances. If this is the case, the measure should also be capable of evaluating a large number of design instances.

Third, a full-fletched case study is conducted on complex biplane iXR systems (cf. Section 3.3(b)). Model validation is achieved by comparing iDSL predictions with real measurements. Furthermore, performance predictions provide insight in the performance of biplane iXR systems with shared hardware in comparison to those with dedicated hardware.

Fourth, we extend iDSL with support for aggregation functions based on generated results, to evaluate designs on different aspects. These aggregation functions can also be intuitively visualized in, among others, trade-off graphs.

Fifth, we extend the latency breakdown chart (of Figure 4.5) with colors and introduce two trade-off graphs (see Figure 5.8 and 5.9), which can be used to compare designs on two aspects.

Outline The remainder of this chapter is organized as follows. In Section 5.2, the extended high-level-performance model of iDSL is introduced. Section 5.3 shows how iDSL transforms into Modest, including the formal definition of eCDF predictions. Section 5.4 conveys the evaluation of iDSL models. Section 5.5 presents the predicted results for many designs and compares them to measurements for validation. Finally, Section 5.6 concludes this chapter.

5.2 The high-level performance model

This section illustrates how iDSL is put into practice by modeling the Image Processing part of a *biplane* iXR system in the iDSL language. For this purpose, we use the six concepts of the iDSL language (as explained in Section 4.2), as follows. Section 5.2.1 and 5.2.2 contain the process, Section 5.2.3 the resource, Section 5.2.4 the system, Section 5.2.5 the scenario, Section 5.2.6 the measure, and Section 5.2.6 the study.

5.2.1 The high-level Process with loads based on eCDFs

Table 5.1 shows the iDSL process of Image Processing (IP, [183]) for a biplane system (cf. Figure 3.3(b)). It consists of two functions “Noise_reduction" and “Refine", which in turn decompose into six and three atomic tasks, respectively. On top of that, IP is enclosed by a Mutual Exclusion (Mutex) that prevents

Table 5.1: The code of an iDSL process with loads based on eCDFs

```
ProcessModel Image_Processing mutex { seq {
    seq Noise_reduction {
        atom Pre_processing load
            eCDF from file "512_monoplane#Preprocessing"
        atom Basic load
            eCDF from file "512_monoplane#Basic"
        atom Decompose1 load
            eCDF from file "512_monoplane#Decompose"
        atom Spatial_noise_reduction load
            eCDF from file "512_monoplane#Spatial"
        atom Temporal_noise_reduction load
            eCDF from file "512_monoplane#Temporal"
        atom Compose1 load
            eCDF from file "512_monoplane#Compose"
    }
    seq Refine {
        atom Decompose2 load
            eCDF from file "512_monoplane#Decompose"
        atom Refine_step load
            eCDF from file "512_monoplane#Refine"
        atom Compose2 load
            eCDF from file "512_monoplane#Compose"
    }
} }
```

multiple process instances from being present in the process at a time. Hence, at most one process instance is processed at all times. This reduces the complexity of the application, i.e., the whole process is treated as one atomic task, but may come at the cost of decreased resource utilizations and increased process latencies.

In the iDSL process of the previous chapter (see Section 4.2.1), atomic tasks have a load that is either constant or dynamically determined via a uniform distribution. Although this provides a great deal of expressiveness, it is insufficiently capable of capturing the loads as measured on real biplane systems, viz., loads adhering to arbitrary distributions. To this end, the iDSL process is extended to support loads that are eCDFs, which are discrete distributions that are generated from a finite set of measurements. Here, the Empirical Distribution Function (EDF, [5]) is used to transform a number of measurements into a distribution. In the extended iDSL language (cf. Table 5.1), we specify the load

Table 5.2: The code of an iDSL process with loads based on eCDF ratios

```
Section Process
    ProcessModel Image_Processing mutex { seq {
        seq Noise_reduction {
            atom Pre_processing load product {
                eCDF from file "512_monoplane#Preprocessing",
                eCDF call dsi_cdf }
            atom Basic load product {
                eCDF from file "512_monoplane#Basic",
                eCDF call dsi_cdf }
            atom Decompose1 load product {
                eCDF from file "512_monoplane#Decompose",
                eCDF call dsi_cdf }
            atom Spatial_noise_reduction load product {
                eCDF from file "512_monoplane#Spatial",
                eCDF call dsi_cdf }
            atom Temporal_noise_reduction load product {
                eCDF from file "512_monoplane#Temporal",
                eCDF call dsi_cdf }
            atom Compose1 load product {
                eCDF from file "512_monoplane#Compose",
                eCDF call dsi_cdf }
        }
        seq Refine {
            atom Decompose2 load product {
                eCDF from file "512_monoplane#Decompose",
                eCDF call dsi_cdf }
            atom Refine_step load product {
                eCDF from file "512_monoplane#Refine",
                eCDF call dsi_cdf }
            atom Compose2 load product {
                eCDF from file "512_monoplane#Compose",
                eCDF call dsi_cdf
            }
        }
    } }
```

of an atomic task in a biplane system as a reference to an external file (e.g., “512_monoplane#Preprocessing”) containing measurements for a given design (i.e., resolution and mode) and specific atomic task.

Table 5.3: The code of the ratio distribution *dsi_cdf* that compensates resolution and mode of the current design instance

```
Abstract CDF dsi_cdf product {
    select dspace("resolution"){
        "512":      quotient { eCDF from file "512_monoplane#sum"
                               eCDF from file "512_monoplane#sum" }
        "1024":     quotient { eCDF from file "1024_monoplane#sum"
                               eCDF from file "512_monoplane#sum" }
        "2048":     quotient { eCDF from file "2048_monoplane#sum"
                               eCDF from file "512_monoplane#sum" }
    }
    select dspace("mode"){
        "monoplane": quotient { eCDF from file "512_monoplane#sum"
                               eCDF from file "512_monoplane#sum" }
        "biplane":   quotient { eCDF from file "512_biplane#sum"
                               eCDF from file "512_monoplane#sum" }
    }
}
```

5.2.2 The high-level Process with loads of eCDF ratios

In Section 5.2.1, the process load is extended to support eCDF distributions, which increases the model expressivity. However, the extension does not scale well when the number of designs grows large, viz., it requires measurements to be performed for each design. In practice, this is not only a very costly but often also an infeasible procedure, viz., each system to perform measurements on, for a given design, has often yet to be built.

For this purpose, the iDSL process (of Table 5.1) is extended (see Table 5.2 and 5.3) with a mechanism to predict eCDFs for different designs via extrapolation, based on a small number of eCDFs. This mechanism is carefully explained in Section 5.3.2. It uses the product and quotient of eCDFs, which are, respectively, defined as follows:

$$q(p) = r(p) \cdot s(p) \qquad\qquad t(p) = \frac{u(p)}{v(p)}$$

where $p \in [0 : 1]$ is the probability, eCDF q the product of eCDFs r and s, and ratio CDF t the quotient of eCDFs u and v.

Note that the quotient of two eCDFs is not necessarily an eCDF. Namely, if the denominator eCDF increases more than the numerator eCDF at some point, the

quotient of these eCDFs decreases and violates the non-decreasing property of an eCDF. Contrarily, one can proof that a product of multiple eCDFs always yields an eCDF, which we leave as an exercise for the reader.

In the following, we compute multiplier "dsi_cdf" of Table 5.3, which compensates for the selected resolution and mode. It consists of a product of quotients of eCDFs. For illustration, we assume the resolution to be 1024^2 pixels and the mode to be *biplane*. Hence, the second quotient eCDF is selected for both *dspace* operators, as follows. In the first case, the eCDF for resolution 1024^2 pixels is divided by the eCDF of 512^2 pixels (the default resolution), as plotted in Figure 5.4(iii)). In the second case, the eCDF for mode biplane is divided by the eCDF for mode monoplane (the default mode), as plotted in Figure 5.4(ii)). Finally, to compensate for both the increase in resolution and mode, the product of both these quotients is taken.

5.2.3 The high-level Resource

In Table 5.4, we define a system Biplane_PC with one resource CPU. The rate of the CPU is 1, so that we can directly inject time measurements into the process (of Section 5.2.1 and 5.2.2) as if they are loads. Namely, the execution time of a process is the quotient of the corresponding load and rate. This means that the time equals the load, if and only if the rate is 1.

To enable FIFO scheduling in the iDSL system (cf. Section 5.2.4), resource CPU is equipped with a buffer. The buffer size of the CPU, the number of images the resource can temporarily store besides the image that is undergoing processing, is b images, where b is the buffer. This buffer can vary for different design instances, since it is a dimension of the design space as defined the study (see Section 5.2.7). In Section 5.3.3, the semantics of the buffer are provided.

Table 5.4: The code of an iDSL resource

```
Section Resource
    ResourceModel Biplane_PC decomp{
        atom CPU
            rate 1
            buffersize dspace("buffer")
    }
```

Table 5.5: The code of an iDSL system

```
Section System
    Service Image_Processing_Service
        Process Image_Processing
        Resource Biplane_PC
        Mapping assign {
            (Pre_processing,CPU)
            (Basic,CPU)
            (Decompose1,CPU)
            (Spatial_noise_reduction,CPU)
            (Temporal_noise_reduction,CPU)
            (Compose1,CPU)
            (Decompose2,CPU)
            (Refine_step,CPU)
            (Compose2,CPU)
        } scheduling policy { (CPU, FIFO) }
```

5.2.4 The high-level System

In Table 5.5, we model the system with one service IP (instead of two) to reduce complexity and to facilitate the prediction of eCDFs in which mode is a design dimension (of Section 5.2.2). The model combines the just defined process (of Section 5.2.2), resource (of Section 5.2.3), and a mapping that maps all processes, inevitably, to the only resource CPU, in a FIFO manner.

Despite modeling only one service, it is still possible to evaluate the performance of both the services in reality, because the system is symmetric. However, we do need to attribute for the concurrency that takes place in "biplane" mode.

The iDSL process (see Table 5.2 and Section 5.2.2) has been extended for this; the eCDF of each atomic process (see Table 5.1) is multiplied with a ratio distribution "dsi_cdf". Ratio distribution "dsi_cdf" (cf. Table 5.3) is the product of two ratio eCDFs of which the second one depends on the selected mode. Namely, in case of "biplane", it uses the quotient of the execution times of biplane and monoplane as a multiplier, e.g., as in Figure 5.4(ii). In case of "monoplane", it is 1.

5.2.5 The high-level Scenario

The scenario (in Table 5.6) prescribes that images arrive f times per second (or 1000 μs) with fixed inter-arrival times, where f is the frame-rate. This frame-

Table 5.6: The code of an iDSL scenario

```
Section Scenario
    Scenario BiPlane_Image_Processing_run
        ServiceRequest Image_Processing_Service
            at time 0, (1000/dspace("framerate")), ...
```

Table 5.7: The code of an iDSL measure

```
Section Measure
  Measure
    ServiceResponse times
      using 1 runs of 1000 ServiceRequests
```

rate can vary for different design instances, since it is a dimension of the design space that is defined the study (see Section 5.2.7).

5.2.6 The high-level Measure

In Table 5.7, the measure “ServiceResponse times”, which is based on the evaluation technique simulation, yields in one run latencies of 1000 images along with the utilization of the CPU.

5.2.7 The high-level Study

A study encompasses a selection of systematically chosen scenarios whose evaluations should lead to an increased insight in the system’s characteristics. To describe a large number of similar scenarios, the system designer can make use of so-called design spaces, as follows.

A design space consists of a number of design space dimensions, which in turn have a number of possible values. A design space can be represented as a set of design instances, viz., this set of design instances is the n-ary Cartesian product of the n design space dimensions of the design space. By definition, a design instance provides a unique valuation for the n design space dimensions.

The system designer can make use of a design space by adding so-called variation points to the model, which refer to a design space dimension in the design space. These references serve as placeholders and are valued using a design instance, viz., the valuation of this design instance for the design dimension

Table 5.8: The code of an iDSL study

```
Section Study
  Scenario BiPlane_Image_Processing_run
    DesignSpace
      ("resolution"     {"512" "1024" "2048"})
      ("mode"           {"monoplane" "biplane"})
      ("framerate"      {"5"})
      ("buffer"         {"0"})
    DesignSpace
      ("resolution"     {"2048"})
      ("mode"           {"biplane"})
      ("framerate"      {"5"})
      ("buffer"         {"0" "1" "2" "3" "4" "5" "6"})
    DesignSpace
      ("resolution"     {"2048"})
      ("mode"           {"biplane"})
      ("framerate"      {"1" "2" "3" "4" "5" "6" "7" "8"
                         "9" "10" "11" "12" "13" "14" "15"
                         "16" "17" "18" "19" "20" "21" "22"
                         "23" "24" "25" "26" "27" "28" "29"})
      ("buffer"         {"0"})
```

referred to. Hence, by repeatedly valuing these placeholders for many design instances, many different system and scenario models are created.

The iDSL model so far (cf. Table 5.1 till 5.7) repeatedly contains the *dspace* operator to refer to a particular design space dimension, viz., the "resolution" and "mode" dimension in the iDSL process (of Table 5.3), the "buffer" dimension in the iDSL resource (in Table 5.4), and the "framerate" dimension in the iDSL scenario (in Table 5.6). Consequently, the design space must contain these four dimensions so that the resulting design instances can provide a valuation for each dimension. iDSL ensures that the system designer defines the needed dimensions via input validation (part of Figure 4.4).

In the study (in Table 5.8), we define three design spaces that serve three different purposes, as follows. The first design space (top) is used to compare the performance of monoplane and biplane systems for three different resolutions, viz. 512^2, 1024^2 and 2048^2 pixels. The frame-rate and buffer size are constant at 5 and 0, respectively. The quaternary Cartesian product of these four dimensions leads to six design instances (see Table 5.9(a)).

The second design space (middle) is used to measure the effect of the buffer size (ranging from 0 to 6) on the number of timed-out images and image latency.

Table 5.9: The resulting design instances of an iDSL study

(a) The performance of an iXR system

$(512, monoplane, 5, 0)$	$(1024, monoplane, 5, 0)$	$(2048, monoplane, 5, 0)$
$(512, biplane, 5, 0)$	$(1024, biplane, 5, 0)$	$(2048, biplane, 5, 0)$

(b) The time-out/latency trade-off of an iXR system

$(2048, biplane, 5, 0)$	$(2048, biplane, 5, 1)$	$(2048, biplane, 5, 2)$
$(2048, biplane, 5, 3)$	$(2048, biplane, 5, 4)$	$(2048, biplane, 5, 5)$
$(2048, biplane, 5, 6)$		

(c) The time-out/frame-rate trade-off of an iXR system

$(2048, biplane, 1, 0)$	$(2048, biplane, 2, 0)$	$(2048, biplane, 3, 0)$
$(2048, biplane, 4, 0)$	$(2048, biplane, 5, 0)$	$(2048, biplane, 6, 0)$
$(2048, biplane, 7, 0)$	$(2048, biplane, 8, 0)$	$(2048, biplane, 9, 0)$
	...	
$(2048, biplane, 24, 0)$	$(2048, biplane, 25, 0)$	$(2048, biplane, 26, 0)$
$(2048, biplane, 27, 0)$	$(2048, biplane, 28, 0)$	$(2048, biplane, 29, 0)$

The other three dimensions are constant. Note that buffers can reduce the number of timed-outs at the price of an increased latency due to waiting times. The resolution is 2048^2 pixels and the mode *biplane* to investigate the most resource demanding system. This design space yields seven design instances (as shown Table 5.9(b)).

The third design space (bottom) is used to measure the effect of the frame-rate (ranging from 1 to 29) on the number of timed-out images, which are positively correlated. The three other variables are constant. Again, the chosen resolution is 2048^2 pixels and the mode *biplane*. This design space yields 29 designs (see Table 5.9(c)).

5.2.8 The high-level Study with design space constraints

In Section 5.2.7, we specified a study comprising three design spaces. Additionally, we here allow for another and sometimes more convenient way for defining design spaces in iDSL, viz., limiting the design space by means of constraints. A constraint is a boolean expression extended with references to designs space dimensions and values. Moreover, a constraint comes with arithmetic opera-

Table 5.10: The code of an iDSL study, semantically equivalent to Table 5.8, but defined alternatively using constraints

```
Section Study
  Scenario BiPlane_Image_Processing_run
    DesignSpace
      constraint((dspace("resolution") < "2048") ->
        ((dspace("framerate") eq "5") &&
        (dspace("buffer") eq "0")))

      constraint((dspace("buffer") > "0") ->
        ((dspace("framerate") eq "5") &&
        ((dspace("resolution") eq "2048") &&
        (dspace("mode") eq "biplane"))))

      constraint(neg((dspace("framerate") eq "5")) ->
        ((dspace("buffer") eq "0") &&
        ((dspace("resolution") eq "2048") &&
            (dspace("mode") eq "biplane"))))

      ("resolution"     {"512" "1024" "2048"})
      ("mode"           {"monoplane" "biplane"})
      ("framerate"      {"1" "2" "3" "4" "5" "6" "7" "8"
                         "9" "10" "11" "12" "13" "14" "15"
                         "16" "17" "18" "19" "20" "21" "22"
                         "23" "24" "25" "26" "27" "28" "29"})
      ("buffer"         {"0" "1" "2" "3" "4" "5" "6"})
```

tors to compare and compute with these dimensions and values. In practice, a constraint can be the result of, among others, legal requirements, market regulations, ethical principles, and (implicit) knowledge of the system designer. For instance, the system designer might know by experience or intuition that an iXR system with resolution 2048^2 pixels does not perform well at frame-rates greater than 30, without necessarily needing to have hard evidence for this proposition.

For illustration, Table 5.10 shows a semantically-equivalent representation of the previously presented study (of Table 5.8). We observe that it only contains one design space instead of three previously, viz., this design space is a superset of the three design spaces (of Table 5.8), limited by three constraints, to yield exactly the same design instances. The three constraints are directly related to the three design spaces (of Table 5.8), as follows. The first constraint states that designs with a resolution smaller than 2048^2 pixels, must have a frame-rate of 5

and a buffer of 0, as with the first design space. The second constraint states that for designs with a buffer greater than 0, the frame-rate must be 5, the resolution 2048^2 pixels, and the mode *biplane*, as with second design space. Finally, the third constraint applies to designs with a frame-rate different from 5, which are required to have a buffer of 0, resolution of 2048^2, and mode *biplane*, thereby covering the third design space.

5.3 The underlying performance model

In this section, we provide semantics to the iDSL model of biplane systems (of Section 5.2) by specifying how this model transforms into Modest code. We provide transformations for a Process with loads based on eCDFs in Section 5.3.1, Process with loads based on ratio eCDFs in Section 5.3.2, Resource in Section 5.3.3, System (in Section 5.3.4, Scenario in Section 5.3.5, Measure in Section 5.3.6, and Study in Section 5.3.7.

5.3.1 The underlying Process with loads based on eCDFs

In Section 5.2.1, an iDSL process with loads based on eCDFs (in Table 5.1) was provided. In this section, we show in four steps: (i) how measurements are performed on a real system; (ii) how measurements are grouped into execution times; (iii) how execution times are turned into eCDFs; and, (iv) how eCDFs measurements are injected in the iDSL model.

Measuring activities on a real system Measurements on embedded systems are typically performed by executing real program code augmented with stopwatches, during a so-called execution run. Stopwatches administer the starting and ending times of functions that run on different resources. Here, we consider iXR systems that loop in cycles and perform a sequence of n image processing operations $(f_1, f_2, ..., f_n)$ on m parallel resources $(r_1, r_2, ..., r_m)$. Measurements lead to activities $Act : Res \times Cycle \times Time \times Time \times Func$ that specify a resource that performs a given function, in a certain cycle, during a time interval. In Figure 5.1, this is visualized in a so-called Gannt-chart [214].

The system designer requires an iXR system to meet two properties: (i) resources process only one operation for one image at a time, which reduces complexity but comes at the price of a reduced utilization; and (ii) iXR systems adhere to a strict FIFO scheduling policy to preserve the image order. Combined, these two properties ensure non-overlapping functions.

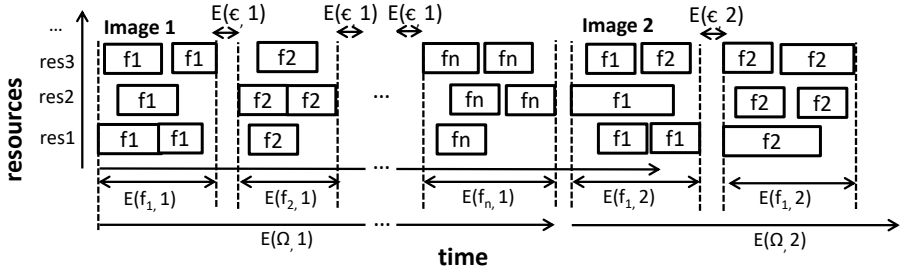


Figure 5.1: Activities displayed in a Gantt-chart: The time is on the X-axis and corresponding resources are on the Y-axis. Activities E are grouped into execution times. Activities form rectangles labelled with the performed function.

Grouping activities into execution times To reduce the model complexity, we combine activities that perform the same functionality, in the same cycle, but on different resources, into one execution time; formally:

$$E_f(c) = \max\{t_2 \mid (r_i, c, t_1, t_2, f) \in Act\} \; - \; \min\{t_3 \mid (r_j, c, t_3, t_4, f) \in Act\}, \qquad (5.1)$$

where f is a function, c the cycle, r_i and r_j resources, t_1 and t_3 starting times, and t_2 and t_4 ending times. Note that execution time $E_f(c)$ may include time during which all resources idled, which can be between function, i.e., the occurrences of $E(\epsilon, n)$ in Figure 5.1, or within a specific function (not shown in the figure). They can be the result of, among others, executing code without stopwatches, or a resource waiting for another resource. Either way, this idle time is attributed to $E_f(c)$ to avoid the underestimation of execution times. Finally, $E_\Omega(c)$ represents the overall execution time; formally:

$$E_\Omega(c) = \max\{t_2 \mid (r_i, c, t_1, t_2, f_i) \in Act\} - \min\{t_3 \mid (r_i, c, t_3, t_4, f_j) \in Act\}. \qquad (5.2)$$

Combining activities leads to an abstraction on both functions and resources.

Using execution times to estimate eCDFs We now *estimate* eCDFs that summarize execution times for different functions. We group the execution times for function f in an array, where we delete the first j samples from $j + k$ measured cycles to eliminate initial transient behaviour. In order to chose a suitable truncation point j, we use the Conway rule [40] because of its simplicity. We

define j as the smallest integer, for each function f, such that the j^{th} execution time is neither the minimum nor the maximum of all preceding samples:

$$\min(E_f(j+1), ..., E_f(j+k)) \neq E_f(j+1) \neq \max(E_f(j+2), ..., E_f(j+1)).$$

This results in array X_f with $|X_f| = k$ elements, where $X_f(i)$, with $1 \leq i \leq |X_f|$, denotes the i^{th} element of X_f:

$$X_f = (E_f(j+1), E_f(j+2), ..., E_f(j+k)). \tag{5.3}$$

Now, let X_f^* be a numerically-sorted permutation of X_f, such that $X_f^*(i) \leq X_f^*(j)$, for all $i \leq j$. Clearly, $|X_f^*| = |X_f| = k$ and again, $X_f^*(i)$ with $1 \leq i \leq |X_f^*|$ denotes the i^{th} element of X_f^*.

In the following, we define the eCDF function e_f and its inverse e_f^{-1} based on X_f^*, for all functions f. The eCDF function $e_f(v) : \mathbb{R} \to [0:1]$ is a discrete, nondecreasing function that returns the probability that a random variable has a value less than or equal to v. It is defined, for each function f, using the widespread EDF [64], as follows:

$$e_f(v) = \frac{1}{k} \sum_{i=1}^{k} \mathbf{1}\{X_f^*(i) \leq v\}, \tag{5.4}$$

where $\mathbf{1}$ is the usual indicator function.

In Figure 5.2, we present an example plot of e_f, based on k values, consisting of $|X_f^*| + 1$ horizontal lines, viz., one for each of the cumulative probabilities $(0, q, 2q, \cdots, 1)$. It can be seen that $e_f(v) = \frac{1}{|X_f^*|} = q$, for $X_f^*(1) \leq v < X_f^*(2)$.

The inverse eCDF function $e_f^{-1} : [0:1] \to \mathbb{R}$ is used to draw samples in line with distribution $e_f(v)$, when simulating. Due to the discontinuities, e_f is not invertible. We resolve this by rounding each probability p to the next higher probability p' for which $e_f^{-1}(p')$ is defined (represented by the vertical dotted lines in Figure 5.2). Thus, $e_f^{-1}(p)$ returns for each $p \in [0:1]$ a value v, as follows:

$$e_f^{-1}(p) = \begin{cases} X_f^*(1), & \text{if } p = 0, \\ X_f^*(\lceil |X_f|\, p \rceil), & \text{if } 0 < p \leq 1. \end{cases} \tag{5.5}$$

This inverse eCDF $e_f^{-1}(p)$ can be used as a basis for the inverse transformation

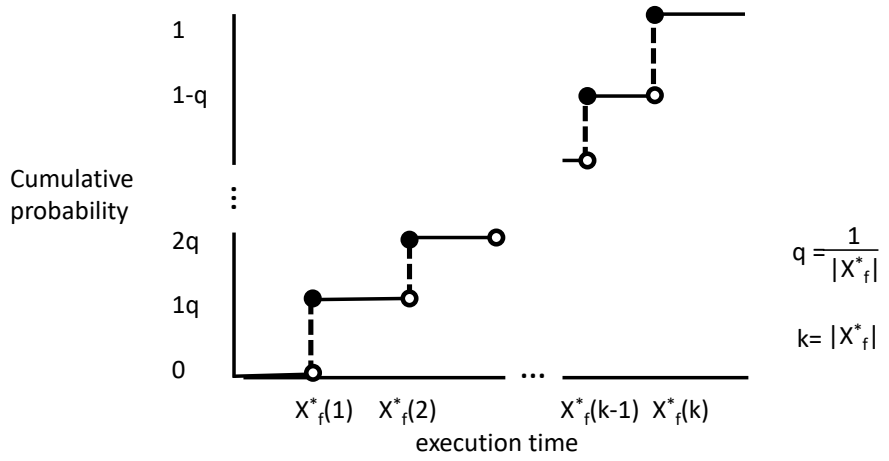


Figure 5.2: The empirical distribution function and its inverse, both based on k samples. They are used to determine the probability that a random variable is below a certain value, and for sampling, respectively. The function shows for each execution time v (X-axis) its cumulative probability p (Y-axis).

method [46], which is a method to generate sample numbers at random using a CDF and a random number generator. Due to the above definition, only actual samples are returned, i.e., neither interpolation nor extrapolation are exercised. It is therefore recommended to use a reasonable amount of measurements.

Injecting the measurements in an iDSL process Finally, we inject the measurements in the iDSL process to calibrate the model, i.e., making the execution times the model generates closely reflect the actually measured execution times.

The measurements (as defined in Equation 5.3) are injected into the iDSL model via a so-called model transformation that transforms the current iDSL model into a semantically equivalent iDSL model. During this transformation, each atomic process that has a load of kind *eCDF from file* is replaced by a probabilistic alternative (palt) in which the weight of each alternative equals 1, and the load corresponds to the value of one of the measurements, in line with the EDF (see Equation 5.5). E.g., Table 5.11 shows the transformation result for process “Pre_processing”, based on actually measured values $1, 4, \ldots, 1$ and 2.

Next, the iDSL process is transformed into Modest code as in the previous chapter (see Section 4.3.1). This transformation does not need to be extended,

Table 5.11: The code of a transformed iDSL process free from eCDFs

```
palt Pre_processing_ecdf{
    1 atom Pre_processing load 1
    1 atom Pre_processing load 4
    ...
    1 atom Pre_processing load 1
    1 atom Pre_processing load 2
}
```

because the new iDSL process is free of *eCDF from file* statements. For the sake of avoiding repetition, we do not include the result of this transformation to Modest here.

5.3.2 The underlying Process with loads of eCDF ratios

In Section 5.2.2, we provide an iDSL process with loads based on ratio CDFs (see Table 5.2 and 5.3). In this section, we show how eCDFs are predicted for the complete design space to reduce the need for costly measurements, as well as how these predictions are injected in the iDSL model, respectively.

Predicting eCDFs for the complete design space We now *predict* eCDFs for different designs choices. Formally, a Design Space has n dimensions, each comprising a set of designs alternatives $dim_i = \{val_1, val_2, ..., val_{m_i}\}$, for $1 \leq i \leq n$. The Design Space Model $DSM : dim_1 \times dim_2 \times ... \times dim_n$ is then the n-ary Cartesian product over all dimensions. A Design Space Instance DSI, also called a “design" or “design instance", provides a unique assignment of values to all dimensions: $\overline{x} = (x_1, x_2, ..., x_n)$, where each entry $x_i \in dim_i$ represents the respective design choice for dimension i.

For the sake of simplicity, $Q_{\overline{x}}$ denotes an inverse eCDF e_f^{-1} that is based on a set of measurements of an execution run for design $\overline{x}$. Additionally, $Q_{\overline{x}}(p)$ denotes a sample drawn from $Q_{\overline{x}}$, for probability p.

Clearly, the number of designs can grow large, viz., exponentially with the number of design dimensions and the number of possible values for each dimension. This makes it costly and infeasible to perform measurements for all possible designs. Hence, we *predict* inverse eCDFs based on other inverse eCDFs without additional measurements, as follows. We carefully select a *base design* $\bar{b}$ to serve as basis for all eCDF predictions, i.e., $\bar{b}$ is a design that performs well so that its execution times mostly comprise service time and no queueing time.

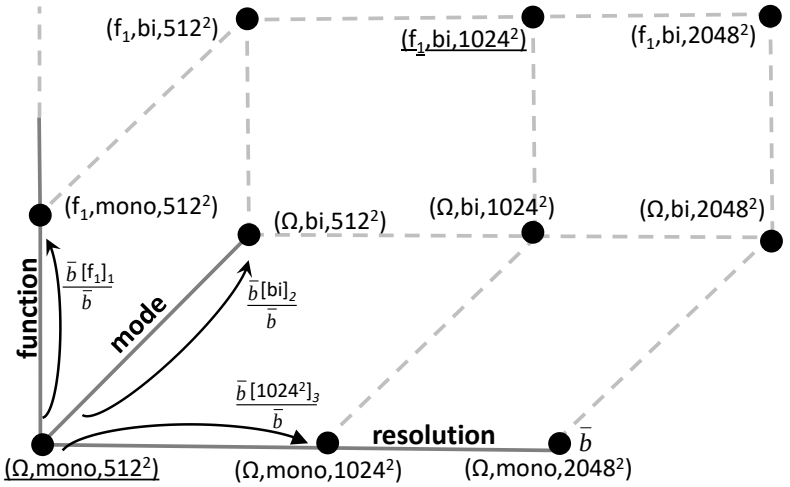


Figure 5.3: A geometric interpretation of a 3D Design Space Model; each spatial dimension represents a design space dimension. Each point in 3D-space represents a Design Space Instance by assigning a value to each dimension. An arrow depicts a ratio between two Design Space Instances.

Hence, $Q_{\bar{b}}$ denotes an inverse eCDF e_f^{-1} that is based on a set of measurements of an execution run for design $\bar{b}$.

Next, we define set $\hat{Q}$ that comprises all inverse eCDFs that need to be acquired through measurements. Set $\hat{Q}$ contains the eCDF corresponding to design $\bar{b}$ and all neighbour designs of $\bar{b}$ that differ in exactly one dimension. Formally, we specify $\hat{Q}$ as a union over all dimensions, as follows:

$$\hat{Q} = \bigcup_{i=1}^{n} \{Q_{\bar{b}[v_i]_i} \mid v_i \in dim_i\}, \tag{5.6}$$

where i is the dimension number, and $\bar{b}[v_i]_i = (b_1, b_2, ..., b_{i-1}, v_i, b_{i+1}, ..., b_n)$.

Now, let $\bar{t}$ be the design for which the inverse eCDF has to be predicted. We assume that all n design dimensions are independent. As we will see below, this assumption does fairly well in the cases we have addressed so far.

Using only inverse eCDFs in $\hat{Q}$, we specify the product of n ratios that each

compensate for the difference between $\bar{b}$ and $\bar{t}$ in exactly one dimension:

$$R(p) = \prod_{i=1}^{n} \frac{Q_{\bar{b}[t_i]_i}(p)}{Q_{\bar{b}}(p)}, \tag{5.7}$$

where $\bar{t}$=(t_1, t_2, ..., t_n), p the probability, and n the number of dimensions.

Measuring all eCDFs in a design space with n dimensions and maximally v values per dimension requires $|DSM| = \mathbb{O}(v^n)$ measurements, while the prediction approach only requires $|\hat{Q}| = \mathbb{O}(vn)$ measurements. Predicting eCDFs is particularly efficient for many dimensions, e.g., for 5 dimensions having 5 values each, prediction requires only 25 out of 3125 (0.8%) eCDFs to be measured.

We illustrate eCDF prediction on an iXR machine with the following three design dimensions (cf. Section 3.1.2): (i) the IP function, which is either f_1, f_2,..., f_n, or Ω (the sum of all functions); (ii) the mode is either *mono*(plane) for one imaging chain, or *bi*(plane) for two parallel imaging chains; (iii) the resolution is the number of pixels of the images processed, and is either 512^2, 1024^2 or 2048^2 pixels. Let $\bar{d} = (f_i, m_j, r_k)$ denote design instance $\bar{d}$ with function f_i, mode m_j, and resolution r_k, which we present conveniently in 3D-space (see Figure 5.3). Additionally, $Q_{\bar{d}}$ denotes the inverse eCDF of this particular design.

Concretely, let $\bar{t} = (f_1, bi, 1024^2)$ be the design, for which we predict an inverse eCDF. Let $\bar{b}$=(Ω, *mono*, 512^2) be the selected base design on which this prediction is based. We then require eCDFs based on measurements for design $\bar{b}$ and for designs that differ on one dimension from $\bar{b}$ and are equal to $\bar{t}$ on this dimension, i.e., designs (f_1, *mono*, 512^2), (Ω, *bi*, 512^2) and (Ω, *mono*, 1024^2).

We assume that the three design dimensions are independent. $R(p)$ is then the product of three ratios that each compensate for the difference between design $\hat{b}$ and $\hat{t}$ in one dimension:

$$R(p) = \frac{Q_{f_1,mono,512^2}(p)}{Q_{\bar{b}}(p)} \cdot \frac{Q_{\Omega,bi,512^2}(p)}{Q_{\bar{b}}(p)} \cdot \frac{Q_{\Omega,mono,1024^2}(p)}{Q_{\bar{b}}(p)}. \tag{5.8}$$

The eCDF of the design $Q_{\bar{t}}$ is then predicted as follows: $Q_{\bar{t}}(p) \approx Q_{\bar{b}}(p) \cdot R(p)$, for probabilities $p \in [0:1]$. For validation, we compare $R(p)$ with ratio $Q_{\bar{t}}(p)/Q_{\bar{b}}(p)$, obtained when measuring $Q_{\bar{t}}(p)$ in Figure 5.4, for all probabilities $p \in [0:1]$.

Figure 5.4 shows the three ratio terms of Equation 5.8: (i) $Q_{(f_1,mono,512^2)}$ / $Q_{\bar{b}}$ (dark blue) compares the execution times of function f_1 and Ω. Function f_1

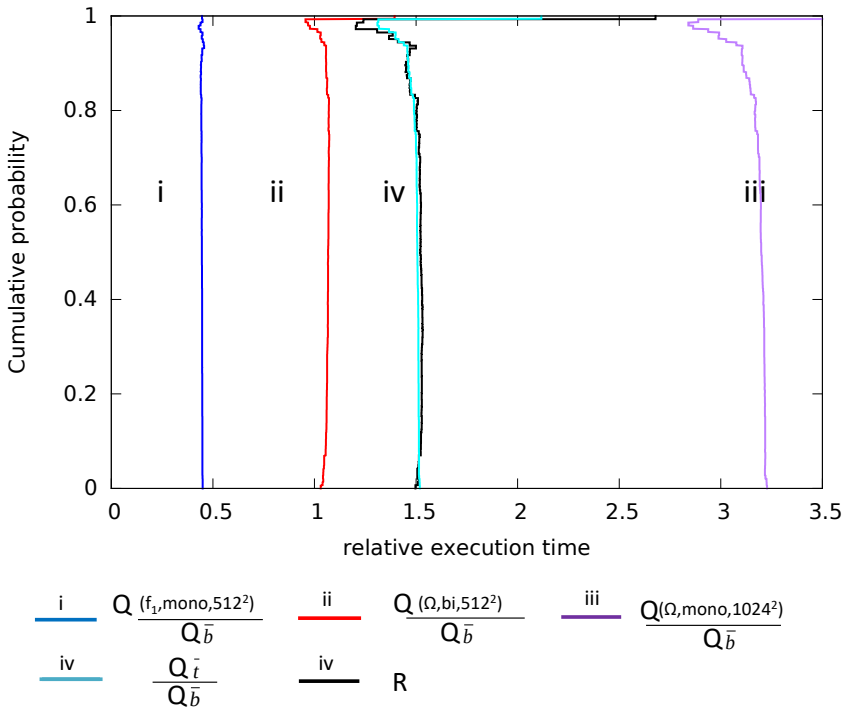


Figure 5.4: Inverse eCDFs with relative execution times, which are the quotient of two eCDFs. On the X-axis, it shows relative execution times, and on the Y-axis, cumulative probabilities. Both axes show ratios and are therefore unitless.

takes about 0.4 of the total execution time; (ii) $Q_{(\Omega,bi,512^2)} \,/\, Q_{\bar{b}}$ (red) compares the performance of a mono and biplane system. Most values are close to 1. Hence, their performance is comparable; (iii) $Q_{(\Omega,mono,1024^2)} \,/\, Q_{\bar{b}}$ (purple) shows the performance effect of a resolution increase from 512^2 to 1024^2 pixels, which is 3.2 for most probabilities p, which is less than the fourfold increase of pixels. Presumably, IP comprises a constant and pixel dependent part, leading to relatively faster processing for larger images. We also see that (iv) $R(p)$ matches its measurement-based counterpart $Q_{\bar{t}}(p)/Q_{\bar{b}}(p)$ well.

The shown graphs are fairly constant for most probabilities p, which indicates that design instances are linearly dependent. However, all graphs display smaller values for probabilities p close to 1. This is in all cases due to inverse eCDF $Q_{\bar{b}}$, which has high execution times for probabilities near 1. Namely, all ratios discussed have $Q_{\bar{b}}$ in their numerator, they consequently display smaller values for the same probabilities. In Section 5.5, we show the results of predicting the performance of designs, using these ratios.

Injecting the predicted measurements in an iDSL process Next, the predicted eCDFs (of Equation 5.8) are injected into the iDSL process. To accommodate different processes per design, iDSL is extended with a process algebra construct *desalt* (for design alternative) to select one out of multiple subprocesses, depending on a design space dimension. In the current case study, a *mode* and *resolution* are selected using nested *desalt* constructs. The desalt con-

Table 5.12: The code of an a transformed iDSL process without ratio eCDFs

```
seq Pre_processing {
    desalt(mode){
        "monoplane":
            desalt(resolution){
                "512":
                    palt Pre_processing_ecdf{
                        ...
                    }
                "1024":
                    palt Pre_processing_ecdf{
                        ...
                    }
                "2048":
                    palt Pre_processing_ecdf{
                        ...
                    }
            }
        "biplane":
            desalt(resolution){
                "512":
                    palt Pre_processing_ecdf{
                        ...
                    }
                "1024":
                    palt Pre_processing_ecdf{
                        ...
                    }
                "2048":
                    palt Pre_processing_ecdf{
                        ...
                    }
            }
    }
}
```

struct has been added to the iDSL to Modest transformation, because it cannot be rewritten in terms of the iDSL language so far.

Contrarily, iDSL code that constains eCDF predictions (see Table 5.2 and 5.3) is transformed into semantically equivalent iDSL code using a number of desalt constructs. For simplicity, we only show the result of the transformation for the "Pre_processing" process in Table 5.12. The process contains two subprocesses (one for each mode) that each contain three subprocesses (one for each resolution), which, in turn, comprise probabilistic alternatives for different latencies. They are precomputed by multiplying and dividing measurements from different files, in line with Equation 5.8. For instance, Table 5.11 gives an impression of what the probabilistic alternatives for different latencies could look like.

Finally, the iDSL process, including *desalt* constructs but free from eCDF constructs, is transformed into Modest code for each design. However, whenever a *desalt* construct is encountered, it is replaced by its proper alternative, i.e., the alternative of which the the design dimension of the desalt and of the current design match. As a consequence, the resulting Modest code is free from *desalt* constructs and predicted measurements. We have not included the Modest models here because many get generated, which are moreover large, viz., thousands lines of code each.

5.3.3 The underlying Resource

Section 5.2.3 provides the iDSL resource of the biplane system (see Table 5.4), which consists of one resource CPU with a rate of 1 and a buffer of size 5.

In comparison to Chapter 4, in which we used a nondeterministic scheduling policy, we now also allow first-in first-out (FIFO) scheduling that is activated using the FIFO keyword in the iDSL mapping (cf. Section 5.2.4 and Table 5.5). It is implemented by extending resource CPU with a buffer, in which process IDs are stored. Consequently, it is possible to keep track of the order in which atomic tasks arrive. In Modest, the buffer is implemented as a data type (cf. Table 5.14) using a list representation under the hood (not shown). It has four functions to respectively add, inspect, remove and count its elements.

In the previous chapter, each iDSL resource resulted in two Modest processes. Here, due to the buffer, each iDSL resource leads to one Modest process and each iDSL process to another Modest process (cf. Table 5.13), as follows.

The resource-dependent Modest process (with prefix "machine_") comprises the following sequence of actions: (i) wait until the buffer contains at least one element; (ii) retrieve the first element from the buffer, which is a process ID;

Table 5.13: The generated Modest code from an iDSL resource

```
buffer CPU_buffer; const int CPU_buffer_size=5;

process machine_CPU(){
  real taskload;
  when urgent ( CPU_buffer.count > 0 )
    tau {= processID = inspect(CPU_buffer),
      CPU_buffer = remove(CPU_buffer) =};
  alt {
    :: machine_CPU_start_1? {= taskload=sync_buffer =};
    :: machine_CPU_start_2? {= taskload=sync_buffer =};
    ...
  };
  delay (taskload / 1 ) tau;
  alt {
    :: when (processID==1) machine_CPU_stop_1!;
    :: when (processID==2) machine_CPU_stop_2!;
    ...
  };
  machine_CPU()
}

process machine_call_Pre_processing(real taskload){
  when urgent (CPU_buffer.count < CPU_buffer_size)
    tau {= process_CPU_buffer =
      add(process_CPU_buffer, 1) =};
  urgent machine_CPU_start_1! {= sync_buffer=taskload =};
  machine_CPU_stop_1?
}
```

Table 5.14: The Modest code definition for the buffer data type

```
datatype buffer = { int count, list? elements };
function buffer add(buffer b, int item) =
  buffer { count: b.count + 1, elements:
  addFirst(b.elements, item) };
function int inspect(buffer b) = getLast(b.elements!);
function buffer remove(buffer b) = buffer { count:
  b.count - 1, elements: removeLast(b.elements!) };
```

(iii) receive the taskload for the process with that ID; (iv) perform the actual delay; (v) notify completion to the process with that ID; and, (vi) self recursion.

The process-dependent Modest processes (with prefix "machine_call") contain the following sequence of actions: (i) wait until the current buffer has space (less than 5 elements in the code of Table 5.13); (ii) add the current process ID to the buffer; (iii) send the taskload to the resource-dependent Modest process; and, (iv) wait for completion of the execution. In Table 5.13, we show the Modest code for atomic task "Pre_processing" with ID 1. Note that a buffer behaves nondeterministically when it is full, because of multiple concurrent processes that might be waiting in step (i). Hence, it is recommended to pick a large buffersize if this behavior is unwanted. Finally, the IDs of processes (all the boldfaced numbers) are statically determined by IDSL.

5.3.4 The underlying System

Section 5.2.4 provided the iDSL system of the biplane system comprising one service (see Table 5.5). It is transformed into Modest code (of Table 4.9) in which all iDSL processes call one Modest process for the CPU. Here, the iDSL system leads to a mapping in which each iDSL process is mapped to an individual Modest process (as in Table 5.15), which is done to support the extension of a resource with a buffer.

5.3.5 The underlying Scenario

In Section 5.2.5, the iDSL scenario of the biplane system is given (see Table 5.6). It consists of one stream of service requests without an initial delay and constant inter-arrival times that are frame-rate dependent. iDSL transforms it into different Modest code for different designs. For instance, we show the designs for which the frame-rate equals 5 (in Table 5.16), i.e., images arrive 200 milliseconds (boldfaced) apart.

5.3.6 The underlying Measure

Section 5.2.6 provides the iDSL scenario of the biplane system (see Table 5.7).

The only measure defined is used to obtain 1000 service response times, based on a single simulation run. This measure resembles the first measure of the previous chapter (see Section 4.2.5 and Table 4.2.5) and leads, therefore, to similar Modest code, viz., the latencies and utilizations are retrieved by extending the already given code with both measurement points and properties. Each

Table 5.15: The generated Modest code from an iDSL system

```
process pre_processing(real taskload){
  machine_call_pre_processing(taskload) }

process basic(real taskload){
  machine_call_basic(taskload) }

process decompose1(real taskload){
  machine_call_decompose1(taskload) }

process spatial_noise_reduction(real taskload){
  machine_call_spatial_noise_reduction(taskload) }

process temporal_noise_reduction(real taskload){
  machine_call_temporal_noise_reduction(taskload) }

process compose1(real taskload){
  machine_call_compose1(taskload) }

process decompose2(real taskload){
  machine_call_decompose2(taskload) }

process refine_step(real taskload){
   machine_call_refine_step(taskload) }

process compose2(real taskload){
  machine_call_compose2(taskload) }
```

(sub)process is enclosed by a stopwatch to register a latency value, which are retrieved via a Modest property. To avoid repetition, we refer to the code of the previous chapter in Table 4.11) for illustration.

5.3.7 The underlying Study

Section 5.2.7 provides the iDSL scenario of the biplane system (see Table 5.8 and Table 5.10). It comprises three design spaces that each contain a number

Table 5.16: The generated Modest code from an iDSL scenario

```
process init_generator_BiPlane_Image_Processing_service()
{
  delay (0)
    generator_BiPlane_Image_Processing_service()
}

process generator_BiPlane_Image_Processing_service(){
  clock c; tau {= c=0 =};
  alt{
    :: generator_image_processing!
    :: delay(1) tau // time-out
  };
  when urgent(c >= (200-0) )
  generator_BiPlane_Image_Processing_service()
}
```

of designs, which in conjunction form a heterogeneous collection of designs.

For each of these designs, Modest code is generated for the study as in the previous chapter (see Table 4.12), after which these designs are individually evaluated as explained next (in Section 5.4).

5.4 Performance model evaluation

We specify our approach as a solution chain as depicted in Figure 5.5, which consists of four consecutive steps. First, pre-processing (in Section 5.4.1) leads to an iDSL model that contains the execution times of each design. Second, during processing (in Section 5.4.2) the actual evaluation using Modest is performed for each measure and design. Third, the first post-processing (in Section 5.4.3) encompasses computing aggregate values for each design, e.g., the average latency of a service. Fourth, the second post-processing step (in Section 5.4.3) leads to trade-off plots to compare different designs on two aspects, as well as lists of designs sorted on one aspect. The iDSL toolbox provides an integral approach towards automating these steps and connecting them seamlessly.

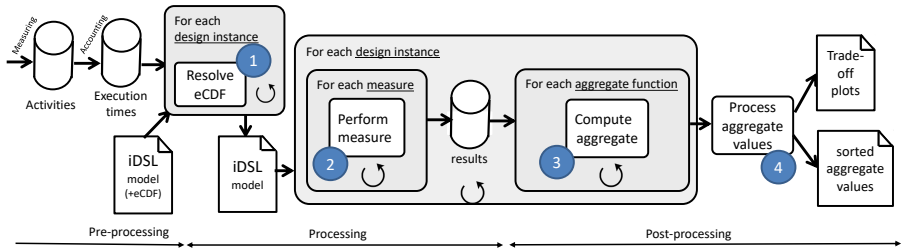


Figure 5.5: The automated iDSL solution chain. An iDSL model and execution times (derived from activities) are used to predict eCDFs, leading to an iDSL model having the predicted eCDFs incorporated in it. For each design, measures are performed and a number of aggregate functions are computed using these measures. Finally, the aggregate values of all design instances are sorted to create trade-off plots.

5.4.1 Pre-processing

During pre-processing, measurements are performed, turned into activities, execution times derived from them via accounting (see Section 5.3.1), which are transformed into eCDFs and injected into an iDSL model.

In detail, measurements are performed for different iXR system configurations and yield large sets of so-called activities for every single design. An activity specifies, for a particular resource and a performed function, the time interval of execution, which is represented as a starting and ending time. Activities are automatically visualized by iDSL in so-called Gantt charts [214], e.g., for human verification and interpretation. They are grouped to obtain total execution times per function and, in turn, aggregated into so-called eCDFs (also in Section 5.3.1).

Next, pre-processing step “Resolve eCDF" performs iDSL model transformations in which eCDF constructs are resolved (see Section 5.3.2). the iDSL model of this chapter (as in Table 5.1 and Figure 5.6) contains nine atomic functions whose probabilistic execution times are individually computed as in Equation 5.8, for each design. In effect, “Resolve eCDF" predicts eCDFs using execution times, followed by a discretization step that turns these eCDFs into finite probabilistic choices. Note that pre-processing makes the approach applicable to complex systems, in line with Objective $\mathcal{I}3.6$.

5.4.2 Processing

During processing, the inverse eCDFs form the starting point, which are all based on measurements and cover all possible designs of interest. Some are used for model validation (explained below) and some are used to predict new, inverse eCDFs. Hence, one is able to reason about the performance of many designs, while relying on only few measurements, in line with Objective $\mathcal{I}1.3$.

Next, the iDSL model is executed to obtain performance results; iDSL performs one simulation per design containing 1000 requests, via the Modest toolset (cf. Table 5.7). This yields results for all designs, meeting Objective $\mathcal{I}3.2$.

5.4.3 Post-processing

During post-processing, aggregated results are processed into aggregated, understandable metrics, facilitating the interpretation of the results, as Objective $\mathcal{I}4.2$ states.

Post-processing step “Compute aggregate" applies, for each design, a number of aggregate functions on the obtained latencies from simulations. Here, the *average*, *maximum* and *median* are the functions of interest (for an example, see Table 5.18), but iDSL also allows for other aggregate functions to be manually defined by the system designer (e.g., Section 6.2.7 and Table 6.7). Concretely, “Compute aggregate" executes each time a simulation run finishes and computes the specified aggregate functions (as in Table 5.18).

Next, step “Process aggregate values" generates trade-off plots [33, 71] that help the system designer with comparing designs by plotting their valuations for two aspects in a 2D-plane (for examples, see Figure 5.8 and 5.9), i.e., they visualize how a gain on one system aspect pays its toll on another, which is in line with Objective $\mathcal{I}4.3$. A trade-off graph is concisely defined in the iDSL measure using two parameters (not shown in this thesis), viz., the aspects of the x-axis and y-axis, respectively. In turn, each aspect is specified using a so-called performance query (to be explained in Section 6.2.7 and 6.2.8). By definition, a design dominates another design when it ranks better on one aspect and is at least a good on the other aspect. Dominated designs are called Pareto suboptimal, all others Pareto optimal.

Finally, step “Process aggregate values" also sorts design instances on each individual aspect. This allows the system designer to compare designs on a particular aspect and see which ones perform best, at a glance.

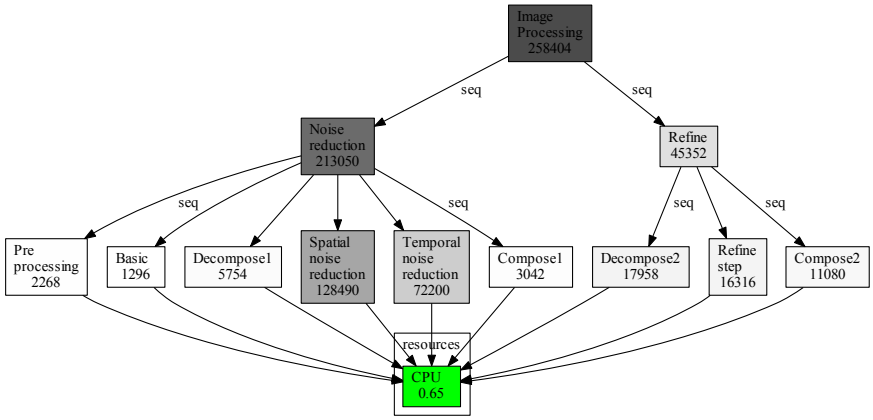


Figure 5.6: Design with resolution "2048^2" and mode "biplane". Service IP contains process IP, which decomposes into a sequential hierarchy of functions. All atomic functions map to resource CPU. In the figure, it shows average latency times (in μs) for each function and the utilization for Resource CPU. Latency demanding functions are dark to be easily pinpointed by the system designer. Resource CPU has a low utilization as represented by the green color. This visual is auto-generated from the iDSL description, which includes the automatic retrieval of performance numbers via performed simulations.

5.5 Performance results

In this section, the results of performance model evaluation (see Section 5.4) are presented for each design in the three design spaces (of Section 5.2.7), as follows. First, Section 5.5.1 addresses the performance of an iXR system for different resolutions and modes. Second, Section 5.5.2 emphasizes the time-out/latency trade-off of an iXR system. Third, Section 5.5.3 conveys the time-out/frame-rate trade-off. Additionally, Section 5.5.4 addresses the validity and applicability of the approach, as implemented in iDSL.

For interpretation, we would like to point out in advance that all the presented results include the design with resolution "2048^2", mode "biplane", buffer "0", and frame-rate "0", because is the only design that is present in both Table 5.9(a), 5.9(b) and 5.9(c). Each result provides a different perspective, as follows. Figure 5.6 conveys average latency times for processes and subprocesses of this design, whereas Figure 5.7(c) (blue) displays a latency distribution for the whole process only. Figure 5.8 shows that this design is one of

the two Pareto optimal ones, while Figure 5.9 conveys that the design is Pareto suboptimal.

5.5.1 The performance of an iXR system

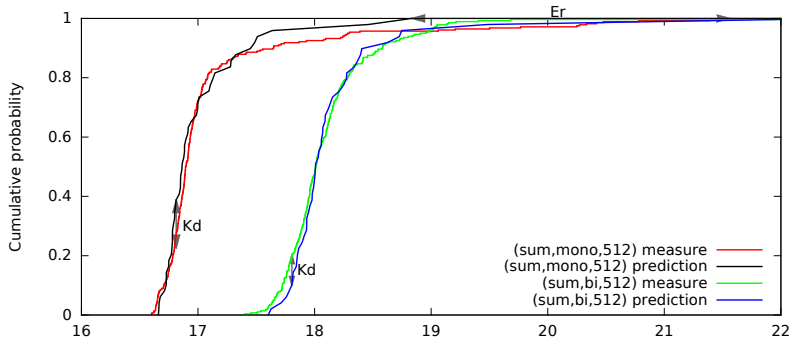
To investigate the performance of an iXR system, we make use of the first design space of the study (in Table 5.8 and 5.10) in which different resolutions and modes are addressed at a constant framerate and buffer. All results were obtained by executing the constructed iDSL model on a PC (AMD A6-3400M, 8Gb RAM) using 32'27" (minutes, seconds) of which predicting eCDFs took 1'48" (6%), simulations 30'13" (91%) and aggregate functions 19" (1%).

We are particularly interested in whether biplane iXR systems on shared hardware perform as good as their monoplane counterparts. Three eCDFs with execution times are used to visually compare the measured performance (in Figure 5.7) of these biplane systems (green) with monoplane ones (red), for resolutions 512^2 (Figure 5.7(a)), 1024^2 (Figure 5.7(b)) and 2048^2 (Figure 5.7(c)). It can be seen that as an effect of sharing hardware, biplane systems perform worse than monoplane ones for image resolutions 512^2 and 1024^2, viz., their average latencies are 6% and 2% higher (as in Table 5.18), respectively. In contrast, biplane systems with an image resolution of 2048^2 perform 9% better than their monoplane counterparts, presumably due to the more powerful hardware the biplane systems are equipped with by design.

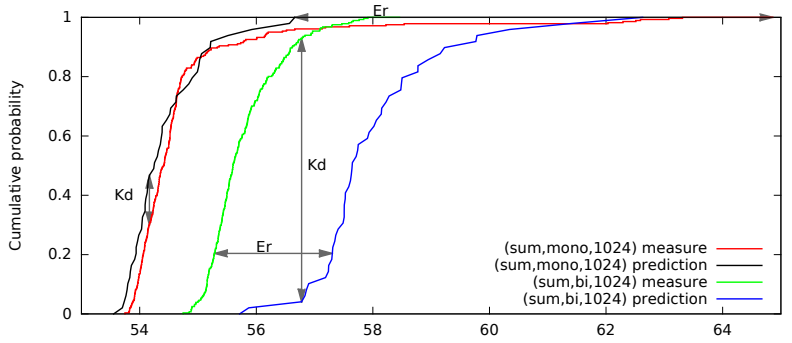
In one go, iDSL also creates latency breakdown charts for *all* designs, including design $(\Omega, 2048, bi)$ that can be found in Figure 5.6. "Spatial noise reduction" and "Temporal noise reduction" (dark gray) are on average the most time consuming functions of IP. iDSL determines the gray value by comparing the subprocess latency with the overall latency. Finally, the CPU utilization of 0.65 is "good" (as green indicates).

5.5.2 The time-out/latency trade-off of an iXR system

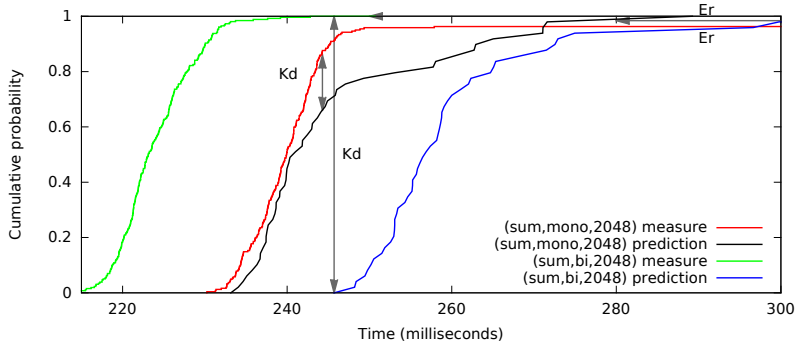
We present a trade-off graph in which the time-out ratio (the relative amount of images rejected by the system due to overuse) and latency are plotted in 2D-space. The graph is based on the second design space of the study (in Table 5.8 and 5.10) and provides insight in how a decrease in the time-out rate implies an increase in the latency, and vice versa.



(a) Resolution 512^2 pixels



(b) Resolution 1024^2 pixels



(c) Resolution 2048^2 pixels

Figure 5.7: Measured and predicted execution times eCDFs for mode monoplane and biplane, which are automatically generated by iDSL

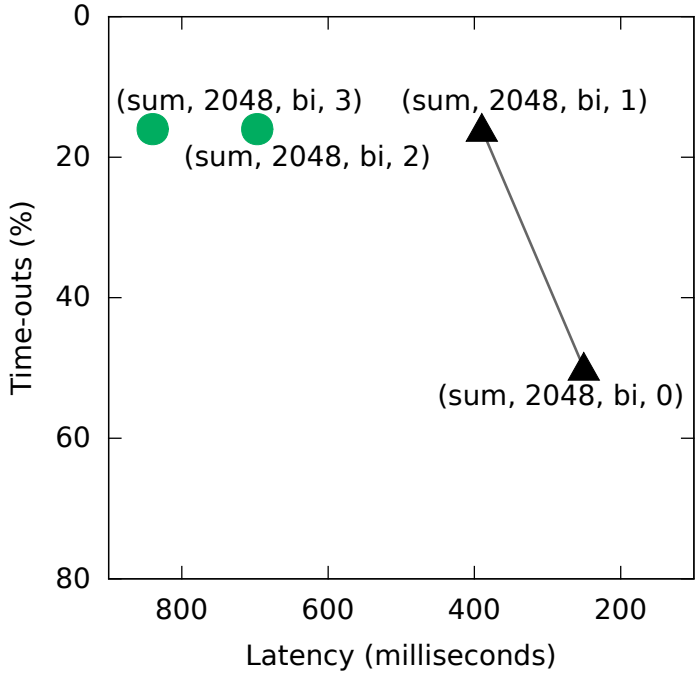


Figure 5.8: A trade-off graph: designs $(\Omega, 2048, bi, b)$ where b is the buffer size affecting both the relative number of time-outs (y-axis) and the average latency (x-axis). Only designs (Ω, 2048, bi, 0) and (Ω, 2048, bi, 1) are Pareto optimal.

Figure 5.8 shows how the buffer size influences both the average latency (x-axis) and the time-out ratio (y-axis), for designs $(\Omega, 2048, bi, b)$ where $b \geq 0$ is the buffer size. Note that both axes are reversed so that preferable designs are plotted on the top-right in the graph, e.g., design $(\Omega, 2048, bi, 1)$ is preferred to $(\Omega, 2048, bi, 2)$. The design with $n = 0$ yields 50% time-outs and a latency of 238ms, whereas the design with $n = 1$ leads to 16% time-outs, but at the price of a higher latency of 367ms. All designs with $n \geq 2$ yield 16% time-outs, but with an ever increasing latency due to queuing time as n increases, making them all Pareto suboptimal.

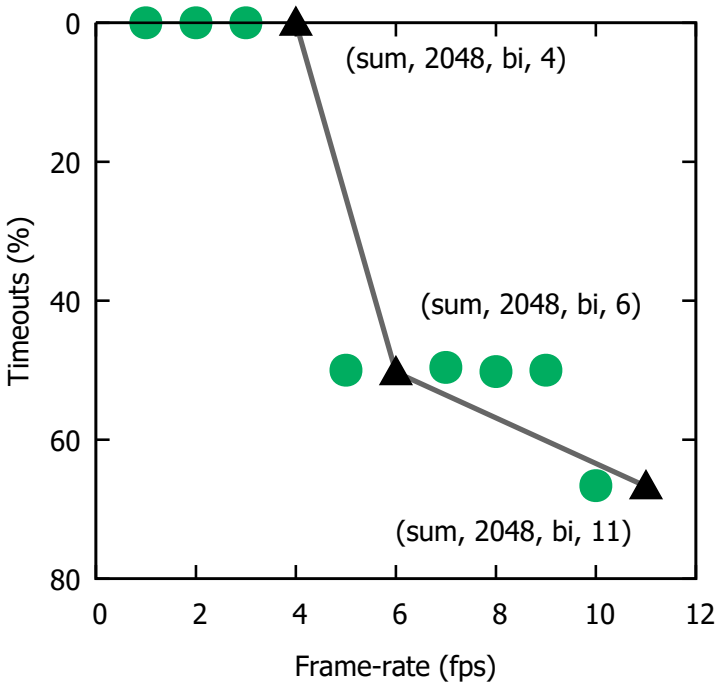


Figure 5.9: A trade-off graph: designs $(\Omega, 2048, bi, f)$ where f is the frame-rate (x-axis) affecting the time-out ratio (y-axis). Pareto optimal designs are depicted by black triangles.

5.5.3 The time-out/frame-rate trade-off of an iXR system

Analogous to the previous trade-off graph, we present a trade-off graph in which the time-out rate and frame-rate are plotted. It is based on the third design space of the study (in Table 5.8 and 5.10).

Figure 5.9 conveys that the frame-rate and the time-out ratio are positively correlated, for designs $(\Omega, 2048, bi, f)$ where f is the frame-rate. Concretely, for $f \leq 5$, no time-outs occur, whereas for $f > 5$ the number of time-outs increases steadily. Hence, increasing the frame-rate above a certain value, viz., ca. 15 frames per second, is discouraged, because the iXR system will simply drop most incoming images instead of processing them.

5.5.4 The validity and applicability of the iDSL model

Finally, we assess whether the predictions reflect reality by comparing the predicted performance (in Figure 5.7) of biplane systems (blue) with the monoplane ones (black). The predictions are consistently higher, which is in line with the difference in the average and median latency of the aggregate metrics (in Table 5.18). This overestimation leads to safe systems, albeit over-dimensioned ones, and originates from eCDF prediction, i.e., it is similar to the ratio for mode between monoplane and biplane: $Q_{(\Omega,bi,512^2)}$ / $Q_{\bar{b}}$ (in Figure 5.4).

In Figure 5.7, distances are shown between eCDFs for both mode monoplane and biplane using two similarity measures for eCDFs, viz., the Kolmogorov distance Kd and the maximum execution ratio Er. Kd is well established in the literature [172] and returns the maximum "vertical distance" between two eCDFs. Kd is defined, as follows:

$$Kd_{m,n} = \sup_{x \in \mathbb{R}} |F_m(x) - F_n(x)|, \tag{5.9}$$

where m and n are eCDFs, and $F_i(x)$ the probability of eCDF i for value x.

Kd is prone to returning high values when the execution times do not show much variation, e.g., a constant execution time of 15.0 compared to a constant (similar) execution time of 15.1 yields a maximum Kd of 1.

To counteract this effect, we propose a second measure, Er, in addition to Kd. Er is inspired by Kd, but returns a "horizontal distance"; these distances are also indicated in the plot in Figure 5.7. Er is normalized using the median values of its arguments, making Er not only symmetric and unitless, but also easy to derive from eCDF graphs. Er is defined, as follows:

$$Er_{m,n} = \frac{\sup_{p \in [0:1]} |G_m(p) - G_n(p)|}{\frac{1}{2}G_m(0.5) + \frac{1}{2}G_n(0.5)}, \tag{5.10}$$

where m and n are eCDFs, and $G_i(p)$ the value of eCDF i for probability p.

Table 5.17 shows outcomes for Kd and Er. It shows the maximum distance p and execution time x it occurred for Kd, and the maximum time ratio at which p occurred for Er. Kd is generally low, i.e., most of its values are below 0.16. However, for design $(\Omega, 1024, bi)$ and $(\Omega, 2048, bi)$, Kd is high, viz., 0.86 and 1, respectively. Figure 5.7(b)) and 5.7(c)) convey that these high Kd values are the result of overestimating predictions.

Table 5.17: Comparing measured and predicted eCDFs for six designs, using two similarity functions: Kd (of Equation 5.9), where x is the execution time with the greatest difference in probabilities. Er (of Equation 5.10), where p is the probability with the greatest relative difference in execution times.

Design	Kd	x	Er	p
$(\Omega, 512, mono)$	0.13	16.8 ms	0.27	1.00
$(\Omega, 512, bi)$	0.08	17.8 ms	0.35	1.00
$(\Omega, 1024, mono)$	0.22	54.0 ms	0.14	1.00
$(\Omega, 1024, bi)$	0.86	56.8 ms	0.04	0.20
$(\Omega, 2048, mono)$	0.15	245.3 ms	0.50	0.98
$(\Omega, 2048, bi)$	1.00	250.3 ms	0.24	0.98

Table 5.18: For three aggregate functions, the predicted (P) and measured (M) outcomes (in milliseconds) and their relative difference Δ, for six designs.

	Average Latency			Maximum Latency			Median Latency		
Design	P	M	Δ	P	M	Δ	P	M	Δ
(Ω,512,mono)	17	17	0%	18	23	-22%	17	17	0%
(Ω,512,bi)	18	18	0%	23	29	-21%	18	18	0%
(Ω,1024,mono)	54	55	-2%	57	65	-12%	54	54	0%
(Ω,1024,bi)	58	56	4%	60	58	3%	58	56	4%
(Ω,2048,mono)	243	245	-1%	298	396	-25%	239	240	0%
(Ω,2048,bi)	259	224	16%	298	251	19%	255	223	14%

Additionally, Table 5.18 shows the measured and predicted outcomes of the aggregated functions. We consider designs with a high value for Kd, viz., $(\Omega, 1024, bi)$ and $(\Omega, 2048, bi)$, and observe that predictions structurally lead to higher values than measurements. For design $(\Omega, 1024, bi)$, predictions have an average latency that is 4% higher, a maximum latency that is 3% higher, and a median latency that is 4% higher than measurements. For design $(\Omega, 2048, bi)$, these values are 16%, 19%, and 14%, respectively. These differences can be attributed to the relative efficiency gain that occurs when the resolution and mode are increased simultaneously, which eCDF prediction as presented in this chapter does not account for.

5.6 Conclusion

In this chapter, we extended iDSL (of Chapter 4), a language and toolbox for performance prediction of Medical Imaging Systems (MISs, [51, 164]), to support the prediction of unseen eCDFs based on other measured eCDFs, aggregate functions, and trade-off graphs, as follows.

In a case study, a model of biplane iXR systems was created that we calibrated using eCDFs based on a few real measurements (c.f. Indicator $\mathcal{I}1.3$). This model enabled us to investigate the performance effect of having biplane iXR systems with shared hardware, opposed to dedicated hardware. Measurements indicate that these biplane systems perform as good as monoplane ones, but predictions yield more conservative results with higher latencies.

We have evaluted many designs (c.f. Indicator $\mathcal{I}3.6$)) and derived aggregated metrics for each one of them (c.f. Indicator $\mathcal{I}4.2$).

In the extended approach, iDSL presents its predictions visually (c.f. Indicator $\mathcal{I}4.3$), e.g., iDSL generates latency breakdown charts in color, for each design, that visualize the process structure, highlight the time consuming processes and resource utilizations, at one glance.

Moreover, iDSL also generates two trade-off graphs for which many design instances were explored and evaluated (c.f. Indicator [$\mathcal{I}3.2$). These trade-off graphs provide the following practical insights to the system designer: (i) the buffer size of a biplane iXR system operating at an image resolution of 2048^2 pixels should not be larger than one image; and (ii) frame-rates exceeding eight images per second on the same system yield many lost images.

In model validation, the predicted outcomes generally reflect reality well. They are, however, conservative for high resolution biplane systems, because they involve two subsequent prediction steps, namely for both the resolution and mode.

In short, the extended performance evaluation approach successfully fulfills the five objectives: (i) it relies on few costly measurements (Indicator $\mathcal{I}1.3$); (ii) evaluates many designs (Indicator $\mathcal{I}3.2$); (iii) automatically generates aggregated metrics (Indicator $\mathcal{I}4.2$); (iv) visualizes them using trade-off graphs and latency breakdown chats (Indicator $\mathcal{I}4.3$); and, (v) evaluates the performance of real complex systems (Indicator $\mathcal{I}3.6$).

CHAPTER 6

Computing Exact Response Time Distributions

This chapter is based on the following publication.

F. van den Berg, J. Hooman, A. Hartmanns, B.R. Haverkort, and A. Remke. Computing Response Time Distributions Using Iterative **[203]** Probabilistic Model Checking. In *Computer Performance Engineering*, volume 9272 of *Lecture Notes in Computer Science*, pages 208–224. Springer, 2015. doi: 10.1007/978-3-319-23267-6_14

6.1 Introduction

Performance prediction early in design is difficult [90]. Discrete-event simulation (see Section 2.2.2) can provide an indication of average response times, but simulation results tend to be averages since rare events are hard to find using this technique [176, 180]. In contrast, worst-case execution time analysis [166] leads to (often non-strict) absolute bounds only. While it can prove that *hard real-time* requirements are met, the computed upper bounds are often pessimistic, leading to costly, over-dimensioned implementations. For many applications, *soft real-time* guarantees are sufficient. The system designer only needs to know whether deadlines are met with a certain (high) *probability*, e.g., the latency value for which, in the long run, 85 % of the service requests complete. These questions can be answered by probabilistic model checking (see Section 2.2.4), in which probabilistic, nondeterministic and timed aspects are combined in one model, leading to precise response time distributions.

In Chapter 4, the iDSL language and tool chain were introduced for evaluating the performance of service-oriented systems and tested on a small example of interventional X-ray (iXR, [158, 159]) system. In Chapter 5, this approach was extended to include, among others, model calibration via measurements. Although model calibration was adequately dealt with, its application significantly increases the model complexity, viz., all atomic processes with a load based on measurements are represented as a probabilistic alternatives construct, with one alternative per measurement. Due to this increase in model complexity, we only used one iDSL measure in Chapter 5 based on discrete-event simulation (see Section 2.2.2). This yields results that are imprecise, because they are obtained using a limited amount of simulation runs of limited lengths and therefore vary much depending on how probabilistic and nondeterministic choices are resolved.

In this chapter, we introduce a new measure that returns response time distributions. It is based on iterative probabilistic model checking (see Section 2.2.4), which guarantees precise results as the model is traversed exhaustively. However, the measure is, especially when applied to complex models, prone to the so-called state space explosion problem [36, 37], i.e., the state space does not fit in computer memory and/or takes too much time to explore. We counteract this drawback by introducing two sampling methods for measurements that simplify the model at the cost of some precision, as follows.

1. Changing the model time unit: divide all times in a model by a constant number (and round them), then perform the evaluation, and finally multiply all results by the same constant number again.

2. Nondeterministic time sampling: replace the measurements of each (or a selection of a) atomic process by the smallest nondeterministic time range that contains all measurements.

In a case study, the new measure is tested on a model that is hard to analyze, viz., it is calibrated using measurements (as in Chapter 5), of so-called monoplane iXR system (cf. Chapter 3 and Figure 3.3(a)). These systems deliver a continuous stream of images to a surgeon that operates on a patient. Low latency response times of images are a must for proper hand-eye coordination [112], i.e., the surgeon must perceive the shown images to be real-time. Nevertheless, a few misses of response deadlines are acceptable, because rare misses only lead to very short degradations in the image quality (also called glitches) that cannot be noticed by the surgeon.

Related work There is relatively little work available to evaluate response time *distributions* for generic system models. The tagged customer approach [79] is a numerical method to compute the response time distribution for open queuing networks. 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 evaluation tool (HIT, [18, 199]) (see Appendix B.7) supports the model-based evaluation of computing system performance, via highly structured models that are based on functional hierarchies and modularization, as with the Y-chart philosophy [119]. The models can be analyzed using multiple techniques, depending on the HIT model class, but response time distributions are not specifically supported.

Additionally, Modular Performance Analysis with Real-Time Calculus (see Appendix B.3) is based on the Network Calculus [27] and computes hard lower and upper bounds using event streams. Hence, these three approaches do not return *precise* response time distributions.

Finally, Platform Independent Petri net Editor (see Appendix B.12) is capable of returning passage times using HYpergraph-based Distributed Response-time Analyser (HYDRA, [28,48], which requires an input PMNL model and a network of commodity PCs.

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* [107]. Typical performance evaluation approaches build on fully stochastic formalisms, such as Continuous-Time Markov Chains (CTMCs) or stochastic Petri nets. However, our examples require a mixture of deterministic timing with probabilistic effects and concurrency; we also want 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, [15, 129]) as the semantic basis of our models.

The analysis of PTA (of Section 2.1.5) is supported by a number of tools including Probabilistic Symbolic Model Checker (PRISM, [128, 165]) and the Modest toolset [88]. We use the latter due to its relatively high-level input language and the ability to not only perform model checking using the included MCSTA tool but also simulation using the MODES simulator.

Approach This chapter presents an automated approach that allows computing response time distributions using iterative probabilistic model checking, which builds on Chapter 4 and the model calibration feature of Chapter 5. The high-level performance model is presented (in Section 6.2), followed by its translation to the underlying performance model (in Section 6.3). Then, the performance model evaluation (in Section 6.4) specifies iterative probabilistic model checking as an evaluation technique. This leads to results (in Section 6.5), i.e., precise response time distributions. Finally, Section 6.6 concludes this chapter.

6.2 The high-level performance model

We provide a general outline of the monoplane iXR system (cf. Chapter 3 and Figure 3.3(a)) of the case study, as follows. An iXR system provides a continuous stream of high-quality images to support a surgeon that operates a patient, which are based on X-ray beams and undergo Image Processing (IP, [183]). The IP service decomposes into a pipeline of twelve image processing steps that are all performed on a resource CPU via a FIFO scheduling policy. Service requests are incoming, unprocessed images that arrive with fixed inter-arrival times, 10 per second in the case study. A monoplane iXR system processes one stream of service requests. The iXR system responds to these requests with processed images, after some latency for processing. This latency should be low enough to make the surgeon perceive the image stream to be real-time. To assess the impact of image quality on performance, we consider two configurations with image resolutions of 512^2 and 1024^2 pixels, respectively.

In the remainder of this section, iDSL is used to specify the iXR system of the case study above. For this purpose, the six concepts of the iDSL language (as explained in Section 4.2) are explicitly represented, as follows. Section 6.2.1 and 6.2.2 contain the Process, Section 6.2.3 the Resource, Section 6.2.4 the system, Section 6.2.5 the Scenario, Section 6.2.6 the Measure which is extended with performance queries in Section 6.2.7 and 6.2.8, and finally Section 6.2.9 the Study.

Table 6.1: The code of an iDSL process with abstract loads

```
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
        }
    }
```

6.2.1 The high-level Process

Process “Image_processing” (in Table 6.1) specifies how images are processed, viz., via two high-level operations “Noise_reduction” and “Refine”, which in turn decompose in a sequential pipeline of twelve image operations.

Each image operation has its independent load (an amount of work), which is specified via an abstraction mechanism, viz., a so-called “call” construct is used that refers to an abstract load that is specified elsewhere in the iDSL code. Note that these abstract loads lead to an elegantly small definition of the iDSL process. In Section 6.2.2, three instantiations of abstract loads are defined.

6.2.2 The high-level Process with sampling methods

Next, we implement the abstract loads of the process (Section 6.2.1 and Figure 6.1). Note that Probabilistic Model Checking (of Section 2.2.4) is defined as a means to evaluate the model (in Section 6.2.6) in this chapter, which is a new iDSL measure. Although this measure delivers precise results, it is also known

to be prone to the state space explosion problem [36,37]. Hence, it requires the size of state space to be limited by all means.

As already demonstrated in Chapter 5, the iDSL process contributes significantly to the size of the state space for two reasons. First, we use many, viz., 300, latency measurements for each function, which are performed on a real iXR system to calibrate the model. Second, the previous chapter conveys that measurements transform into complex process algebra structures (see Table 5.12).

Therefore, the iDSL process is simplified by making use of one out of three sampling methods that simplify the measurements by approximating them, viz., *uniform sampling*, *changing the model time unit*, and *nondeterministic time sampling*. The selection of sampling method may affect a number of factors, such as the computational time and memory needed, and the quality of the resulting model (cf. Section 2.3.2). The sampling methods are implemented as an abstract load for each image operation (see Table 6.2). In what follows, we explain how each sampling method works. For illustration, we provide an abstract load "preproc" for each sampling method, as referred to by "Pre_processing" in the iDSL process (of Section 6.1).

Uniform sampling First, the abstract load "preproc" is implemented for uniform sampling (see Table 6.2(a)). The load is defined for two image resolutions, viz., 512^2 and 1024^2 pixels, which is the only dimension of the design space as defined in the study (in Section 6.2.9). The *dspace* operator yields the selection of the right set of measurements, given the image resolution of the current design. In turn, *Uniform from file* refers to an external file containing the measurements. In uniform sampling, each measurement has an equal probability to be sampled. Hence, it can lead to much complexity, especially when the measurements show much variety. Namely, only similar measurements are grouped together.

Changing the model time unit Second, the abstract load "preproc" is implemented for changing the model time unit (see Table 6.2(b)). Again, the load is defined for two images resolutions, as with uniform sampling. When changing the model time unit, however, measurement values are divided by a given constant number and rounded to make the model simpler at the price of some precision. After evaluation, the results are multiplied by the same constants again to compensate for the division. For instance, we divide by 250 for resolution 512^2, and 800 for 1024^2 pixels in the case study. These numbers have been found empirically to work well, i.e., smaller numbers induced a state space ex-

Table 6.2: The code of abstract load “preproc", part of the iDSL process

(a) Uniform sampling

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

(b) Changing the model time unit

```
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
}
```

(c) Nondeterministic time sampling

```
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
}
```

plosion, whereas larger numbers led to very inaccurate results. In Chapter 7, this approach is extended to automatically obtain these numbers by systematically performing benchmarks (cf. Section 7.5.1), hidden from the system designer.

Nondeterministic time sampling Third, the abstract load “preproc” defines the variable load of “Pre_processing” using nondeterministic time sampling (see Table 6.2(c)). Once more, the load is defined for two images resolutions, as with uniform and abstract sampling. In nondeterministic time sampling, the load is defined as a nondeterministic range $[min : max]$, where min is the minimum of value among all measurements and max the maximum. This means that the frequencies of the measurements are neglected and that the range is strict, i.e., it does not contain a value that is greater or smaller than all measurements. In the next chapter (see Section 7.5.1, the clustering of measurements), we extend this way of sampling to one in which we combine a number of probabilistically selected nondeterministic segments.

Table 6.3: The code of an iDSL resource

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

We have provided the implementation of image operation “Pre_processing” for three sampling methods. The remaining eleven image operations are implemented analogously, viz., only the references to the external file with measurements are different for each image operation. Under the hood, the sampling methods are implemented via model transformations. In Section 6.3.1, we present the iDSL code resulting from these transformations, for each sampling method.

6.2.3 The high-level Resource

The resource “Image_processing_PC” is defined (in Table 6.3). It consists of a CPU with a rate of 1, i.e., it can process 1 unit of load per μs, the chosen time unit of the case study. The rate is chosen 1 to facilitate the injection of time measurements into the process (as already explained in Section 5.2.3). Additionally, the resource is equipped with a buffer of size 10 to enable first-in first-out (FIFO) scheduling in the iDSL system (cf. Table 6.4), as thoroughly explained in Section 5.3.3.

Table 6.4: The code of an iDSL system

```
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) }
```

6.2.4 The high-level System

In Table 6.4, the iDSL system is shown, which comprises one service named "Image_Processing_Service'. This service connects process "Image_Processing" to resource "Image_Processing_PC" by means of a mapping. This mapping prescribes that all twelve image operations are performed on the only resource CPU and that a FIFO scheduling policy is used to resolve concurrency.

6.2.5 The high-level Scenario

The iDSL scenario (in Table 6.5) comprises a scenario "Image_Processing_Run" in which the service (of Section 6.2.4) is invoked without an initial offset and 10 times per second with constant inter-arrival times, i.e., once every 100000 μs, forever.

Table 6.5: The code of an iDSL scenario

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

6.2.6 The high-level Measure

The iDSL measure section specifies the following two measures (see Table 6.6).

First, measure "CDF of ServiceResponseTimes" 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 has a latency below 60 ms. This measure is obtained via model checking, which is generally slower than simulation. However, it conveys different insights, e.g., absolute lower and upper bounds. The evaluation of this measure is explained in detail in Section 6.4.

Second, "ServiceResponseTimes" 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. In this chapter, simulation results are used to validate the results that have been acquired using model checking.

Table 6.6: The code of an iDSL measure

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

6.2.7 The high-level Measure: simple performance queries

In the following, we extend the iDSL measure section (of Section 6.2.6) with relatively simple performance queries, i.e., they do not require much post-processing. The queries will be used to answer the following two performance questions.

$Q1$. 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?

In general, a performance query is either a so-called *utility* or *cost* function. That is, it is a utility function when a higher value is preferable, e.g., the percentage of service requests completed after a given time, and a cost function when a lower value is preferable, e.g., the average latency of a service. A performance query is an arithmetic expression that usually contains one or more references to measures, but may also contain values of design dimensions. Its evaluation yields a real number for each design, obtained by evaluating its arithmetic expression in which references to measures are replaced by actual measurements.

A performance query can be used, among others, to compare designs on an aspect to enable design space exploration [13, 14, 104]. Moreover, two performance queries yield a trade-off plot (as shown in Figure 5.8 and 5.9).

Next, we present two types of performance queries, viz., either based on model checking or simulation, that lead to answers for question $Q1$ and $Q2$.

Model checking-based performance queries We introduce four performance functions that rely on measure "CDF of ServiceResponseTimes", as follows.

1. Function $Q1_{lb}$ returns the minimum latency before which P percent of the service requests of service S completes (cf. Table 6.7(a)).

2. Function $Q1_{ub}$ returns the maximum latency before which P percent of the service requests of service S completes (cf. Table 6.7(a)).

3. Function $Q2_{lb}$ returns the minimum (maximum) percentage of service requests of service S that has a latency below time T (cf. Table 6.7(b)).

4. Function $Q2_{ub}$ returns the maximum percentage of service requests of service S that has a latency below time T (cf. Table 6.7(b)).

Functions $Q1_{lb}$ and $Q2_{lb}$ return minima, and functions $Q1_{ub}$ and $Q2_{ub}$ yield maxima. The functions have a service S parameter, and moreover either a percentage P or a time T parameter.

Table 6.7: Extending the iDSL measure with simple performance queries

(a) Latency times, using model checking

```
Cost Q1a_lb timeAfterPercentageHasFinished
  (Service Image_Processing_Service, Percentage 85, tmin)

Cost Q1a_ub timeAfterPercentageHasFinished
  (Service Image_Processing_Service, Percentage 85, tmax)
```

(b) Latency percentiles, using model checking

```
Utility Q2a_lb percentageBelowTime
  ( Service Image_Processing_Service, Time 55 us, pmin )

Utility Q2a_ub percentageBelowTime
  ( Service Image_Processing_Service, Time 55 us, pmax )
```

(c) Latency times, using simulation

```
Cost Qla_sim percentile 85 of latencies
  ( Service Image_Processing_Service, Run 1, Request 1..100 )
```

(d) Latency percentiles, using simulation

```
Utility Q2a_sim percent below 55 of latencies
  ( Service Image_Processing_Service, Run 1, Request 1..100)
```

In iDSL, these functions are implemented using performance queries with parameters service "Image_Processing_Service" (of Section 6.2.4). For illustration, we select either percentage "85" or time "55" only. In Table 6.7(a), the implementations of $Q1_{lb}$ and $Q1_{ub}$ are provided, whereas $Q2_{lb}$ and $Q2_{ub}$ are implemented in Table 6.7(b).

Simulation based performance queries We introduce two performance functions that rely on simulation-based measure "ServiceResonseTimes", as follows.

1. Function $Q1_{sim}$ returns the latency before which P percent of the service requests of service S complete, based on R simulation runs of Rq requests each (cf. Table 6.7(c)).

2. Function $Q2_{sim}$ 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 (cf. Table 6.7(d)).

Note that functions $Q1_{sim}$ and $Q2_{sim}$ have simulation-specific parameters for the number of simulated runs R and requests Rq per run, respectively. Hence, the system designer can configure how often and long the simulations are performed. Additionally, functions have a service S parameter, and either a percentage P or time T parameter.

Furthermore, we see that model checking based performance queries have distinct maximum and minimum functions; the difference between them is the result of nondeterminism. Simulation based performance queries do not have this distinction, because nondeterminism is resolved ad hoc, during simulation, e.g., using a combination of first-in first-out (FIFO) and as soon as possible (ASAP) scheduling (see also Section 4.4.3).

Again, these functions are implemented using performance queries in iDSL with parameters service "Image_Processing_Service", and either percentage "85" or time "55". Table 6.7(c) implements $Q1_{sim}$ and Table 6.7(d) $Q2_{sim}$.

Finally, iDSL is equipped with input validation for defining functions to ensure the right measures, e.g., simulation, are defined to retrieve them as well as their parameters, e.g., the simulation-based functions (of Table 6.7(c) and 6.7(d)) require the simulation run to be at least of length 100.

6.2.8 The high-level Measure: complex performance queries

Next, we build on Section 6.2.7 by introducing performance queries with a higher degree of complexity, i.e., they require more post-processing to be per-

Table 6.8: Extending the iDSL measure with complex performance queries

(a) The maximum number of pixels processed per second

```
Utility Q3_pixe1_per_sec
  ((dspace("resolution")*dspace("resolution"))/
  minimum of latencies (
      Service Image_Processing_Service, Run 1, Request 1..100))
```

(b) An average latency time based on simulation and model checking

```
Utility Q4_average_latency
  (( average of latencies (
      Service Image_Processing_Service, Run 1, Request 1..100)+
    timeAfterPercentageHasFinished (
      Service Image_Processing_Service, Percentage 50, tmin))+
    timeAfterPercentageHasFinished (
      Service Image_Processing_Service, Percentage 50, tmax))
```

formed. Without going into details, we here just provide two examples of such queries:

$Q1$. What is the maximum number of pixels that can be processed per second?

$Q2$. What is the latency for which a given percentage of the service requests completes, based on both model checking and simulation runs?

We introduce two performance functions (cf. Table 6.8) that rely on simulation-based measure "ServiceResonseTimes" and/or model checking-based measure "CDF of ServiceResponseTimes", as follows.

1. Function $Q3_{pixel\_per\_sec}$ returns the maximum throughput in pixels of service S (cf. Table 6.8(a)).
2. Function $Q4_{average\_latency}$ returns the percentage of service requests of service S that has a latency below time T, based on both R simulation runs of Rq requests each and model checking (cf. Table 6.8(b)).

Function $Q3_{pixel\_per\_sec}$ provides a metric that is independent of the size of the images, viz., it uses the number of images of both design for normalization.

The results (in Table 6.12(c)) convey that the performance for images with 512^2 and 1024^2 can be directly compared, viz., they are of the same magnitude of order, in contrast with comparing just latency times (of Table 6.12(a)).

Function $Q4_{average\_latency}$ is constructed to return an average latency. However, instead of relying on one evaluation technique for this, such as simulation or model checking, it makes use of both. Namely, it is defined as the average of: (i) the average of simulation results, (ii) minimum average latency, and (iii) maximum average latency. The use of multiple sources does not only make the function more robust, but can even yield better results than the individual sources, similar to ensemble learning algorithms in machine learning [217].

6.2.9 The high-level Study

In the iDSL study (see Table 6.9), a one dimensional design space is used to define two design instances of iXR systems that process images of resolution 512^2 and 1024^2 pixels, respectively. By default, iDSL evaluates each one of them individually using the two earlier defined measures (of Table 6.6).

Table 6.9: The code of an iDSL study

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

6.3 The underlying performance model

In Section 6.2, we specified the monoplane iXR system of the case study in the iDSL language. In this section, we provide the semantics of this system via a transformation from iDSL into the underlying Modest code, as follows. Section 6.3.1 covers the Modest Process, Section 6.3.2 the Resource, Section 6.3.3 the System, Section 6.3.4 the Scenario, Section 6.3.5 the Measure, and we conclude with the Study in Section 6.3.6.

6.3.1 The underlying Process

In Section 6.2.1, the iDSL process of the monoplane iXR system (see Table 6.1) has been specified. It contains twelve atomic tasks that execute in sequence, at

its lowest level of abstraction. The loads of the tasks are defined using a "call" construct, which is a reference to an abstract load specified elsewhere in the iDSL code.

In Section 6.2.2, the iDSL process (of Table 6.1) is extended using three different abstract loads that each correspond to a different sampling method. In iDSL, abstract loads are injected into the iDSL process via a model transformation. This leads to a new iDSL process in which atomic processes are free from "call" constructs. We now explain the resulting iDSL code of atomic process "Pre_processing" after this transformation for the three proposed sampling methods (as in Table 6.2), viz., *uniform* sampling, *changing the model time unit*, and *nondeterministic time* sampling.

Uniform sampling iDSL transforms the abstract load of "Pre_processing" for uniform sampling (see Table 6.2(a)) into one without "call" and "from file" constructs. In the following, we only show the result for a resolution of 512^2 pixels, whereas the result for 1024^2 pixels is similar (but not shown).

The result (see Table 6.10(a)) is a probabilistic alternatives (palt) construct of which the alternatives directly relate to different values in the 300 measurements, viz., 130, 131, ..., and 136 μs. On top of that, the alternatives are weighted by how often their value has been observed, in line with the Empirical Distribution Function (EDF, [5]). For instance, the probability for 130 μs is $\frac{8}{300}$, because 8 out of 300 measurements are 130 μs, for 131 μs it is $\frac{67}{300}$, because 67 out of 300 measurements are 131 μs, etc.

The complexity of this sampling method can be expressed in the number of alternatives of the palt (of Table 6.10(a)), i.e., 7 in the example. However, for atomic processes yielding larger measurement value ranges, e.g., the most complex processes "spatial_noise_red" and "temporal_noise_red", we found the range of values to be much larger, resulting into much more alternatives in the palt construct. Clearly, the degree with which this sampling method reduces complexity is, among others, case study specific.

Changing the model time unit To change the model time unit, iDSL transforms the abstract load of "Pre_processing" (see Table 6.2(b)) into one without "call" and "from file" constructs. We only show the result for a resolution of 512^2 pixels, whereas the results for 1024^2 pixels is similar, as follows.

In this particular case, the result (see Table 6.10(b)) is a probabilistic alternatives (palt) construct that consist of only one "alternative" having load 1. Namely, all measurements are have been divided by 250, and rounded to the

Table 6.10: The code of iDSL atomic process “preproc" for three sampling ways

(a) Uniform sampling

```
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 }
```

(b) Changing the model time unit

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

(c) Nondeterministic time sampling

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

nearest integer. For measurements 130, 131, ..., and 136 μs, thist means that all representations are $\frac{130}{250} = 0.52 \approx 1$, $\frac{131}{250} = 0.524 \approx 1$, ..., $\frac{136}{250} = 0.544 \approx 1$. To compensate for this division, results are multiplied by 250 again after evaluation. For more complex processes, e.g., “spatial_noise_red” and “temporal_noise_red”, the resulting palt construct displays more variation.

This divider 250 has been empirically found to work well, i.e., smaller numbers introduced a state space explosion problem [36, 37], whereas larger numbers led to inaccurate results. It can be seen that this sampling method can be used to get rid of an arbitrary amount of complexity. On one hand, selecting an arbitrary high division factor will lead to loads that are 0 only, while on the other hand, selecting a (too) high division factor will jeopardize the accuracy of the results due to excessive rounding. It can also been reasoned that the rounding involves some hard-to-predict coincidence, i.e., if many measurements are (close to) a multiple of the divider, then rounding errors are minimal. In Chapter 7, iDSL is extended to determine the divider in an empirical but fully automated way (cf. Section 7.5.1).

Nondeterministic time sampling iDSL transforms the abstract load of process “Pre_processing” for nondeterministic time sampling (see Table 6.2(c)) into one without “call” and “from file” constructs. In the follow, we show the result for a resolution of 512^2 pixels. The result for 1024^2 pixels is similar.

The result (see Table 6.10(c)) is a probabilistic alternatives (palt) construct that consist of only one "alternative". In contrast to uniform and changing the model time unit, this alternative has a nondeterministic load with range [130 : 136], which contains all measurements and is bounded by the minimum and maximum measurement. Consequently, all measurements 130, 131, ..., and 136 μs are taken into account during evaluation, albeit without considering their respective probabilities. Hence, evaluation leads to a range of possible results, without probabilities.

This sampling method can be used to drastically reduce complexity. It works particularly well when the loads are fairly constant, since the nondeterministic ranges are small then. However, the sampling method eliminates the possibility of knowing how likely latencies of a certain magnitude occur. This might be sufficient for many use cases, but is not enough for the case study on iXR systems. Namely, in the case of have to meet a hand-eye coordination [112] criterium, it is acceptable to have a high latency every now and then as long as most latencies are below a certain value. Contrarily, nondeterministic sampling leads to an absolute maximum latency only and does not convey information about the distribution of the other latencies.

6.3.2 The underlying Resource

Section 6.2.3 provides the iDSL resource of the monoplane iXR system (in Table 6.3). At its lowest level of hierarchy, it consists of one resource CPU, virtually the same as the iDSL resource of the previous chapter (see Table 5.4). Consequently, it is transformed into Modest code that is similar to the code of the previous chapter (see Table 5.13).

6.3.3 The underlying System

Section 6.2.4 provides the iDSL system of the monoplane iXR system (in Table 6.4), which is similar to the iDSL system of the previous chapter (see Table 5.5). Therefore, it is transformed into Modest code that is similar to the code of the previous chapter (see Table 5.15), viz., mapping each atomic function to an underlying function.

6.3.4 The underlying Scenario

Section 6.2.5 provides the iDSL scenario of the monoplane iXR system (in Table 6.5), which is similar to the iDSL scenario of the previous chapter (see Ta-

ble 5.6), However, the period of the incoming requests is different, viz., the period is constant here and varies per design in the previous chapter. Therefore, it is transformed into Modest code that is almost the same as the code of the previous chapter (see Table 5.16), except for the just mentioned period (boldfaced).

6.3.5 The underlying Measure

Section 6.2.6 provides the iDSL measure of the monoplane iXR system (in Table 6.6), which is extended in Section 6.2.7 with performance queries (see Table 6.7). Hence, the iDSL measure contains two measures and a number of performance queries, as follows.

First, measure "ServiceResonseTimes" is based on simulation and has already been introduced (in Chapter 4 and 5). It is solely used in this chapter to validate the obtained results (in Section 6.5), and will no further be addressed in this chapter's remainder.

Second, measure "CDF of ServiceResponseTimes" uses probabilistic model checking and forms the key contribution of this chapter. It leads to a cumulative distribution function (CDF) for each service and does not require parameters in the iDSL language, viz., its results are universal. The measure leads to the generation of a Modest model that adheres to Probabibilistic Timed Automata (PTA, see Section 2.1.4) in a similar way as measure "ServiceResponse absolute times" (of Section 4.3.5) uses Timed Automata (TA, see Section 2.1.3). The Modest model is evaluated via probabilistic model checking (of Section 2.2.4) using MCSTA of the Modest toolset under the hood.

Third, the iDSL measure contains performance queries (of Section 6.2.7 and 6.2.8), which result into a post-processing step of evaluation, which we discuss in Section 6.4.6.

6.3.6 The underlying Study

Section 6.2.9 provides the iDSL study of the monoplane iXR system (in Table 6.9). It contains a 1-dimensional design space in which two design instances are defined with image resolutions 512^2 and 1024^2 pixels, respectively. In iDSL, this leads to the generation of similar Modest code for each design instance that relies on different measurements in the iDSL process.

6.4 Performance model evaluation

In this chapter, a new measure "CDF of ServiceResonseTimes" was defined (in Section 6.2.7 and 6.3.5). The measure relies on (iterative) probabilistic model checking and is used to compute latency response times. Consequently, the measure can be used to answer performance questions, which are represented as performance queries (of Section 6.2.7) in iDSL.

The measure involves the following six steps to be performed in sequence: (i) an iDSL model is transformed into several Modest models that are used to retrieve service latencies (cf. Section 6.4.1); (ii) these latencies are aggregated into one overarching latency per service (cf. Section 6.4.2); (iii) the overarching latency is computed using MCSTA (cf. Section 6.4.4) (iv) Multiple iterations of MCSTA lead probability bounds (cf. Section 6.4.6); (v) these probability bounds are transformed into a set of possible CDFs (cf. Section 6.4.6); and, (vi) performance questions are answered using the set of possible CDFs (cf. Section 6.4.5).

6.4.1 From iDSL queries to Modest

A range of modest models are generated to answer performance queries, viz., for each iDSL model i, each service s within that model, and both the minimum and maximum probability (selected using 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 of Section 6.3. A measure is added to the service that is measured in a given model, i.e., its process is enclosed by stopwatches that record latencies of its service requests. In case of more than one service, only one is measured at a time. Finally, a property to retrieve the minimum or maximum probability ($p_{\min}$ or $p_{\max}$) that a service completes within time t is added. Therefore, 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, as follows:

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

where p is either the minimum or maximum probability (depending on flag f) that service s in iDSL model i completes within time t. In Section 6.4.3, we provide the iDSL implementation of function $\mathcal{M}$ (see Figure 6.2).

6.4.2 Aggregating Latencies of Service Requests

In the previous step, 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 its own latency. Ideally, the average of this infinite stream of latencies is a measure for the performance of the whole service. Put formally:

$$P_{\Omega}(t) = \lim_{k \to \infty} \frac{1}{k} \sum_{n=1}^{k} P_n(t), \tag{6.1}$$

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 cannot be directly computed. That is why we show the following two weighted averages of the latencies that are computable and can thus be used to approximate the measure. For a start, the *arithmetic mean* considers the first N service requests and weighs them equally, as follows:

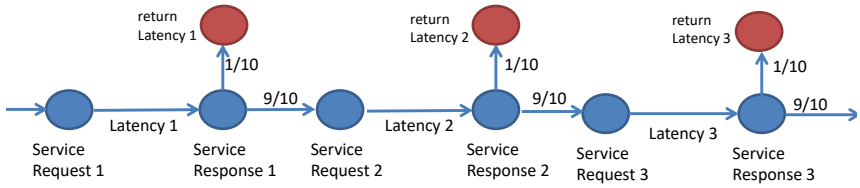
$$P_{\Omega}(t) = \frac{1}{N} \sum_{n=1}^{N} P_n(t), \tag{6.2}$$

where $N \in \mathbb{N}^+$ is the number of service requests considered, e.g., $N = 100$. This arithmetic mean is similar to Equation 6.1 for large values of N. Larger N-values lead to a more complex model but more precise results, and vice versa. However, regardless of magnitude N, there are two drawbacks: (i) it requires a counter to be added to the state in Modest to keep track of the service request number that causes the state space to explode [36, 37]; and (ii) latencies of the $(N+1)^{th}$ service request and later are neglected, which means that a absolute maximum latency can be overlooked.

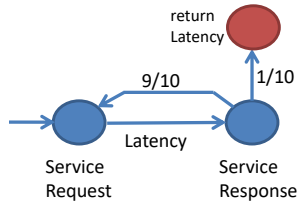
To overcome these two drawbacks, we propose to use the *geometric distribution* [156] to weigh service requests in an exponentially decreasing fashion, as follows:

$$P_{\Omega}(t) = \sum_{n=1}^{\infty} (1-\rho)^{n-1} \, \rho \, P_n(t), \tag{6.3}$$

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



(a) with a service request counter



(b) without a counter

Figure 6.1: Binary probabilistic choices induce the geometric distribution

This geometrically weighted distribution is, again, similar to Equation 6.1 for ρ close to 0. The geometric distribution takes the latencies of all service requests into account. It therefore ensures finding the absolute latencies.

In Modest, the geometric distribution is implemented as a binary probabilistic choice every time a service request completes (as depicted in Figure 6.1(a)): either the currently measured latency is returned, with probability ρ ($\frac{1}{10}$ in the figure), or the next service request is evaluated, with probability $1 - \rho$ ($\frac{9}{10}$ in the figure). Moreover, the geometric distribution is memoryless, i.e., the binary choice does not rely on state information. Hence, it is possible to omit the service request number from the model, leading to a single reoccurring service request (as in Figure 6.1(b)). In the remainder of this chapter, we only consider the geometric distribution with $\rho = \frac{1}{10}$, empirically determined to be a good compromise between state space size and accuracy of results.

Also, Figure 6.1(b) conveys that a lower ρ-value leads to a more complex model but more precise results, and vice versa. Namely, when value ρ is small, state "Service Response" is more likely to be followed by "Service Request" for many times. This does lead to more accurate results though, since a wide array of latencies is considered instead of only a few in the beginning. Regardless of the selected value for ρ, the absolute latencies will be found, since the value for ρ only affects the probabilities between states but not the reachability.

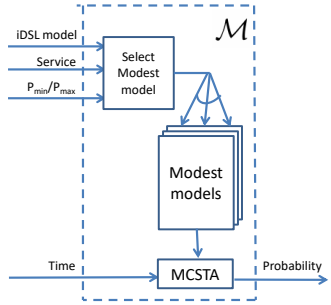


Figure 6.2: Abstract function $\mathcal{M}$ returns the probability a service finishes within a certain time, (given an iDSL model, and minimum or maximum).

6.4.3 Computing the overarching service latency

Under the hood, Modest applies probabilistic model checking (of Section 2.2.4) to evaluate the performance of the model (of Section 6.4.2), which is of type PTA (see Section 2.1.4). Value iteration (see Section 2.2.4) is a key procedure here, viz., it ensures that global minima and maxima latencies are found regardless of which policy is used to resolve nondeterminism.

MCSTA is applied to the just specified Modest models. For the sake of simplicity, we use abstract function $\mathcal{M}$ (of Section 6.4.1) to retrieve a probability:

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

where p is the probability that service s in iDSL model i completes minimally or maximally (p_{minmax}) within time t

iDSL implements $\mathcal{M}(i, s, f, t)$ in an automated manner, via the following three steps (as illustrated in Figure 6.2). First, it generates the Modest model corresponding to iDSL model i, service s and flag f. Second, it runs this Modest model in MCSTA with parameter time t. Third, it returns the result of MCSTA as probability p.

6.4.4 Iterative Model Checking for Probability Bounds

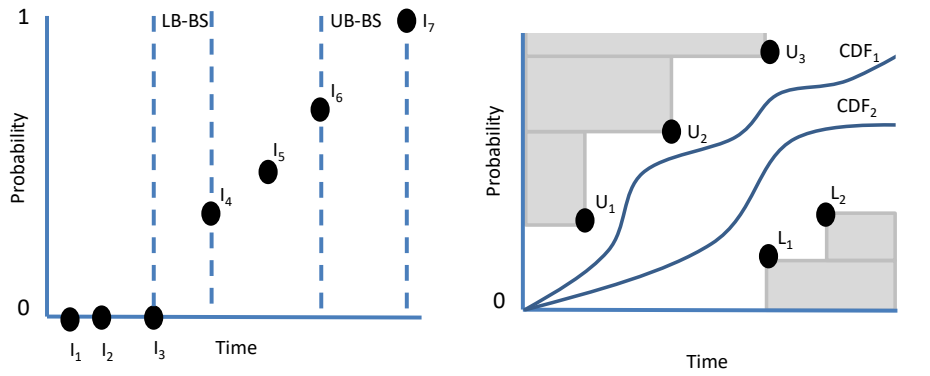
In the previous section, we have illustrated that probabilities are computable for a given latency. Next, we provide an algorithm that uses function $\mathcal{M}$ to compute probability bounds, for a given iDSL model i, service s in this model and a

minimum/maximum bound flag f. This is accomplished by iteratively applying function $\mathcal{M}(i, s, f, t)$ for different values of time t. This procedure comprises three stages, viz., an initial scan, a binary lower & upper bound search, and a brute force computation, as follows.

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 this computation is deterministic, which means m and n are unique values.

Figure 6.3(a) depicts the initial scan graphically. It shows computations i_1, i_2, ..., i_7, with i_7 having a probability of 1. We deduct 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 (see also Section 4.4.4) are applied to ranges $[2^m : 2^{m+1}]$ and $[2^n : 2^{n+1}]$ for the lower and upper bound, respectively.



(a) The initial scan and resulting bounds

(b) The set of possible CDFs in continous space

Figure 6.3: Cumulative Distribution Functions (CDFs) based on function $\mathcal{M}$

In Figure 6.3(a), the areas of the binary searches are labeled *LB-BS* and *UB-BS*, 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 $t \in [lb : ub]$, needs to be determined yet.

Brute force computation. We obtain $\mathcal{M}(i, s, p_{\min/\max}, t)$ for all times $t \in [lb : ub]$. They are computed in parallel 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 resource-consuming duplicate computations. Cache hits particularly include brute force computations that have already been performed while searching for bounds. This cache is possible, because $\mathcal{M}$ is deterministic, i.e., repeating a $\mathcal{M}$ computation with the same parameters will always yield the same result.

6.4.5 Transforming Bounds into a set of possible CDFs

In the following, we investigate how the lower and upper bounds (as obtained in Section 6.4.4) reduce the set of possible CDFs in continuous 2D-space. However, these sets are of infinite size and consequently not countable. To counteract this, we will additionally show how the lower and upper bounds reduce set of possible CDFs in a discrete 2D-space, yielding finitely countable sets that can be computed by iDSL using a tailored algorithm.

Continuous space 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 6.3(b) illustrates how five probabilities (upper bounds U_1, U_2 and U_3, and lower bounds L_1 and L_2) constrain 2D-space. That is, the left-top area of upper bounds and right-bottom area of lower bounds are colored gray. It also shows two example CDFs that respect these bounds, because they do not cross any gray area.

We consider the set of all CDFs that respect these bounds, which encompasses all CDFs that are above all lower bounds and below all upper bounds for all times t. Formally, function $CDF_{all} \colon I \times S \to 2^{\widehat{CDF}}$ returns, where $\widehat{CDF}$ is the universe of all CDFs, given an iDSL model $i \in I$ and service $s \in S$, the set of CDFs that respect the bounds in $\mathcal{M}$:

$$\begin{aligned} CDF_{all}(i,s) = \{\, cdf \in \widehat{CDF} \mid cdf(0) = 0 \\ \wedge\, \forall t\ (\ \mathcal{M}(i,s,p_{\min},t) = p_1 \Rightarrow cdf(t) \geq p_1\ ) \\ \wedge\, \forall t\ (\ \mathcal{M}(i,s,p_{\max},t) = p_2 \Rightarrow cdf(t) \leq p_2\ )\,\} \end{aligned} \tag{6.4}$$

Note that constraint $cdf(0) = 0$ has been added to require all the values to be greater than or equal to 0, viz., we do not consider negative amounts of time.

Discrete space In the following, we illustrate how a bound reduces the set of possible CDFs. For the sake of simplicity, we consider a finite set of all and possible CDFs by assuming a discrete space. Consequently, we can count CDFs using the discipline of combinatorics [198], as follows.

First, we discretize the probability and time dimension to yield a 2D grid (as visualized in Figure 6.4). The use of small units yields more precision but comes at the price of more complex computations. We use probability steps of $\frac{1}{8}$ and time steps of 1. Second, we add two constraints, viz., U_4 and L_3, to limit the upper and lower bound of all CDFs. Namely, the size of the set of unbounded CDFs is always infinite. Implicitly, we also assume point $(0,0)$ to be a constraint, in line with Equation 6.4

Next, given the 2D grid of Figure 6.4 and bounds U_4 and L_3, we conclude that the number of valid CDFs is $\binom{16}{8} = 12870$. This is the number of shortest walks from point $(0,0)$ to $(8,1)$ (in Figure 6.4), which encompasses 8 moves to the right and 8 moves upwards, in any order.

Similarly, we determine the number of CDFs upper bound U_2 prohibits, given bounds U_4 and L_3. For this purpose, we pay attention to the diagonal on the left-top of U_2 with points $(2, \frac{3}{4})$,$(1, \frac{7}{8})$ and $(0,1)$, as highlighted in Figure 6.4). We reason that: (i) CDFs that contain each of these points are not permitted by U_2; (ii) a CDF can maximally contain one of these three points, i.e., they are mutual exclusive due to the non-decreasing property of CDFs; and, (iii) a CDF that is prohibited by U_2 but allowed by U_4 and L_3 must contain one of these three points. These insights combined tell us that the CDFs that U_2 prohibits must contain exactly one of the points $(2, \frac{3}{4})$,$(1, \frac{7}{8})$ and $(0,1)$.

We now determine the number of CDFs U_2 prohibits given U_4 and L_3. The number of CDFs that include point $(2, \frac{3}{4})$ is $\binom{8}{6} * \binom{8}{2} = 784$, viz., the number of shortest walks from $(0,0)$ to $(3, \frac{3}{4})$ multiplied by the number of shortest walks from $(3, \frac{3}{4})$ to $(8,1)$. In a similar fashion, the number of CDFs that include point $(1, \frac{7}{8})$ is $\binom{8}{7} * \binom{8}{1} = 64$, and for point $(0,1)$ it is $\binom{8}{8} * \binom{8}{0} = 1$. From the above we

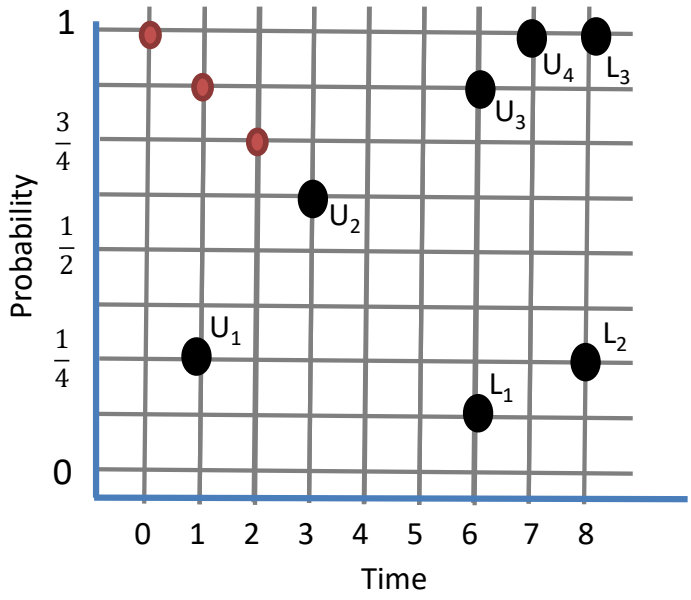


Figure 6.4: Restricted points by U_2 in the discretized set of possible CDFs

conclude that the number of possible CDFs, given upper bound U_2, and bounds U_4 and L_3, is $\sum_{n=0}^{2} \binom{8}{8-n} * \binom{8}{n} = \binom{8}{6} * \binom{8}{2} + \binom{8}{7} * \binom{8}{1} + \binom{8}{8} * \binom{8}{0} = 849$. Hence, upper bound U_2 prohibits $\frac{849}{12870} \approx 6,6\%$ of the total CDFs, given bounds U_4 and L_3.

Lower bounds are dealt with analogously, which means the diagonal on the right-bottom is used instead. The effect of multiple bounds is harder to determine since the sets of CDFs they prohibit may overlap. It is beyond the scope of this thesis.

6.4.6 Answering the Performance Queries using the CDFs

We now use CDF_{all} of Equation 6.4 to answer the performance queries, as follows. Queries of type $Q1$, the minimum (or maximum) time for which a service request completes with probability p, are determined, as follows:

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

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

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

After this, the queries of type Q3 and Q4 (cf. Section 6.2.8) are answered based on the queries of type $Q1$ and $Q2$, in a post-processing step.

6.5 Performance results

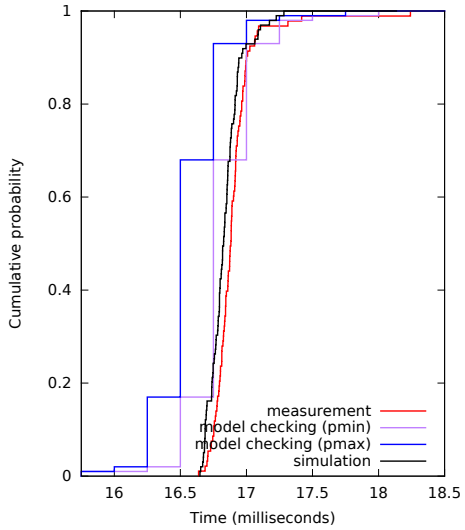
The performance analysis approach (of Section 6.4) is applied to the monoplane iXR system (of Section 6.2). Three experiments are defined (in Section 6.5.1) and their results are compared with simulations and real measurements, in three steps: (i) we present CDFs with latency times (in Section 6.5.2); (ii) we show the execution times and model sizes (in Section 6.5.3); and, (iii) we present the answers to the performance questions for second of the three experiments (in Section 6.5.4).

6.5.1 Three experiments based on sampling methods

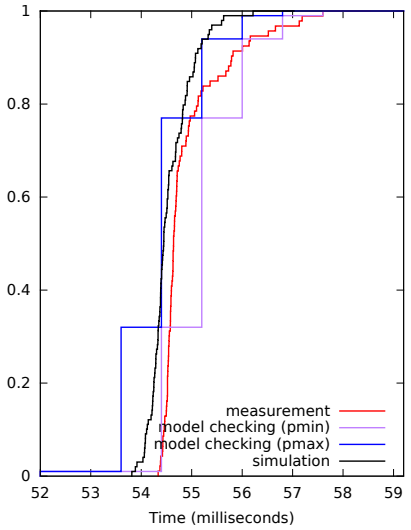
Three experiments are defined, based on and named after the sampling methods (as introduced in Section 6.2.2), as follows. First, experiment "uniform sampling" uses uniform sampling. Running MCSTA leads to a incremental generation of the state space, but runs out of memory and stalls after having generated 38 million states. Hence, we are unable to present results for this experiment and will no further address it in this section anymore. Second, experiment "changing the model time unit" uses changes the model time unit. Finally, experiment "nondeterministic time sampling" uses nondeterministic time sampling, but on an empirically determined subset of the system to reduce complexity, i.e., we only consider the first three image operations instead of all twelve.

6.5.2 CDFs with latencies

In Figure 6.5 and 6.6, latency CDFs are shown, for resolutions 512^2 and 1024^2, and experiment "changing the model time unit" and "nondeterministic time sampling", respectively. In each figure, we show the percentage of service requests that complete, on the Y-axis, within a given latency, on the X-axis. Furthermore, it shows the actual measurements used for calibration (in red), simulated results (in black), and a lower bound result (in blue) and upper bound

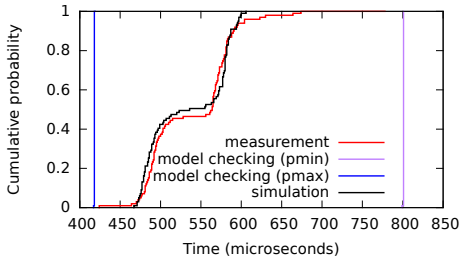


(a) Experiment “changing the model time unit”, Image resolution 512^2 pixels

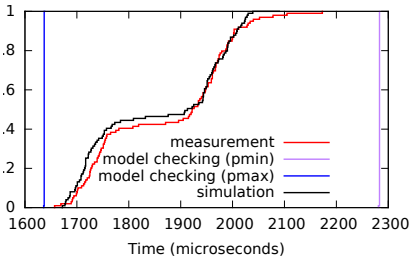


(b) Experiment “changing the model time unit”, Image resolution 1024^2 pixels

Figure 6.5: CDFs with latencies of IP of iXR systems: measurements on a real iXR system, model checking (lower & upper bounds) and simulations.



(a) Experiment “nondeterministic time sampling”, Image Resolution 512^2 pixels



(b) Experiment “nondeterministic time sampling”, Image Resolution 1024^2

Figure 6.6: CDFs with latencies of IP of iXR systems: measurements on a real iXR system, model checking (lower & upper bounds) and simulations.

result (in purple) obtained via model checking. Next, we discuss these results per experiment, followed by a comparison of the experiment results.

Experiment "changing the model time unit" Figure 6.5 conveys, for two image resolutions, that model checking returns lower values than simulation for probabilities close to 0, and higher time values for probabilities close to 1. This indicates that the model checking provides a range of results that is wider than the results obtained via simulations. Hence, the new approach provides more insight in the worst and best case behavior of the system.

Next, we verify if the bounds (in Figure 6.5(a) and 6.5(b)) of model checking encompass all simulated results, as semantically intended. That is, for each latency l, the probability of the lower bound CDF (in blue) should be greater than or equal to the simulated result CDF (in black), and the simulated result CDF should be greater than or equal to the upper bound CDF (in purple).

This is not always the case, as follows. For instance, in Figure 6.5(a) the upper bound CDF (in purple) has probability 0.17 for latency 16.5, while the probability of the simulated result CDF is close to 0. For another instance, in Figure 6.5(b) the lower bound CDF (in blue) has probability 0.93 for latency 55.5, while the probability of the simulated result CDF is almost 1. The two sketched invalidities can be contributed to the time abstraction during which rounding took place.

Experiment "nondeterministic time sampling" Figure 6.6 displays, for two image resolutions, that the computed bounds enclose all measurements, as required. Moreover, simulations display much less variance and quite tighter bounds, because of a combination of the following two reasons. First, when the number and length of the simulation runs are of insufficient size, simulation results will simply not cover a wide variety of results, e.g., rare events will be neglected [180]. Second, when probabilities in the original model are replaced by nondeterministic equivalents, either over- or underestimation will occur when these constructs are encountered during model checking.

Comparing both experiments It can be observed that the results of experiments "changing the model time unit" (see Figure 6.5) lead to less space between the lower and upper bound than the results of (see Figure 6.6), which means that the former experiments provides more information (cf. Section 2.3.2). However, this comes at the price of precision (cf. Section 2.3.2). Namely, not all measurement values are between the lower and upper bound which is the

Table 6.11: Two experiments: execution times for simulations and model checking, the number of states of the Modest model and the number of MCSTA calls.

Experiment	Simulation time (sec)	MC time (hr:min:sec)	Image size (pixels)	states	calls
Experiment 1: time unit	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

case for the latter experiment. In the next chapter, we quantify this loss of information (see Equation 7.1 and 7.2) and loss of precision (see Equation 7.3 and 7.4), and introduce a way to obtain a combination of two sampling methods (see Section 7.1), i.e., “changing the model time unit” and ”nondeterministic time sampling”, in an automated manner.

6.5.3 Execution times and complexities

Table 6.11 shows the execution times and state space sizes for Experiment “changing the model time unit” and “nondeterministic time sampling” on an AMD A6-3400M APU, 8 GB RAM system. 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, while MCSTA is called between 49 and 88 times.

6.5.4 Results of the Performance Queries

In this section, we discuss the answers (see Table 6.12) to the performance queries (cf. Table 6.7) and the advanced performance queries (cf. Table 6.8), for experiment “changing the model time unit” (of Section 6.4.5 and 6.4.6).

In Table 6.12(a), the results of performance queries of type $Q1$ are shown, which are the latency for which a given percentage of service requests completes. We observe, for both image resolutions, that both model checking bounds are lower than simulations for percentile=0, and higher for percentile=100. This indicates that the model checking results display more variety than the simulation results, in line with what Figure 6.5(a) and 6.5(b) show. We assume that simulation is less capable of addressing rare events [176, 180]. For percentiles inbetween, the results are somewhat similar.

Table 6.12: Performance queries: results for two image resolutions.

(a) Type Q1 queries (vertically): three results (in ms) for two image resolutions, viz., for simulation (*sim*), lower (*lb*) and upper (*up*) bounds.

		512*512 pixels			**1024*1024 pixels**		
	percentile	**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

(b) Type Q2 queries (vertically): Three results for two image resolutions, viz., for simulation (*sim*), lower (*lb*) and upper (*up*) bounds.

		512×512 pixels			**1024×1024 pixels**		
	latency	**sim**	**lb**	**ub**	**sim**	**lb**	**ub**
Q2a	55 ms	x	x	x	84%	32%	77%
Q2b	17 ms	91%	91%	96%	x	x	x

(c) Type Q3 and Q4 queries (vertically): results for two image resolutions

	512×512 pixels	**1024×1024 pixels**	**Unit**
Q3	14.5M	18.4M	pixels/second
Q4	16.7	54.7	milli-seconds

In Table 6.12(b), the results of performance queries of type $Q2$ are shown, viz., the percentage of latencies that has a latency below a given value. For latency$=55\,ms$ and resolution$=1024 * 1024$, simulation leads to a value that is slightly higher than the upper bound of model checking. For latency$=17\,ms$ and resolution$=512 * 512$, simulation leads to a value similar to the lower bound of model checking. In conclusion, the model checking bounds can be useful to get an indication of how many service requests meet a certain deadline. However, simulation results can be outside the model checking bounds, albeit slightly, due to a level of imprecision changing the model time unit inherently introduces.

In Table 6.12(c), the results of performance queries of type Q3 and Q4 are displayed. For type Q3, we observe that the number of pixels processed per

second increases when the resolution is higher, similar to the previous chapter. Namely, in Figure 5.4 a fourfold increase of pixels only led to a latency increase between 3.0 and 3.5. For type Q4, the latency times are similar to the ones of $Q1$, as intended.

6.6 Conclusion

We have extended iDSL (of Chapter 4 and 5) with a new evaluation method that retrieves response time distributions, using iterative probabilistic model checking. We have demonstrated that this method can be easily adopted by system designers, viz., they only have to add one line to the iDSL code to specify additional iDSL measure of interest. Also, the measure is backwards compatible with older iDSL models. Hence, the system designer can easily obtain the response time distributions of older iDSL models.

The new iDSL measure is based on probabilistic model checking and therefore prone to the state space explosion problem [36, 37]. This means it is not suitable for complex models, since it explores the model exhaustively. However, we wanted to apply the new measure to a case study on monoplane iXR systems in which model calibration (of Chapter 5) was used to inject measurements in the iDSL. This resulted into a fairly complex model.

Therefore, to enable the use of more complex models, we introduced two new sampling methods to reduce the model complexity, respectively, increasing the model time unit, and replacing probabilism by nondeterminism. These methods have proven to be able to reduce the model complexity, but do compromise the quality of the results and need to be manually added by the system designer. Moreover, the system designer needs to empirically determine a constant value to increase the time unit with. In Chapter 7, these methods to reduce the model complexity are fully automated to not bother the system designer with these.

We validated our approach by (visually) comparing the results of the new measure with simulation results (see Figure 6.5 and 6.6). Not only do the graphs look very similar, despite the use of sampling methods, they also provide a new insight in addition to simulation results. That is, the absolute bounds of the new measure are wider, opposed to simulations which tend to lead to average values, especially when rare events [176, 180] are omnipresent in the model.

CHAPTER 7

Efficiently Computing Exact Response Time Distributions

This chapter is based on the following publication.

[202] F. van den Berg, B.R. Haverkort, and J. Hooman. Efficiently Computing Latency Distributions by Combined Performance Evaluation Techniques. In *Proceedings of the 9th EAI International Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS'15, pages 158–163. ICST, 2015. doi: 10.4108/eai.14-12-2015.2262725

7.1 Introduction

In Chapter 6, we introduced an exact measure to compute response time distributions for service systems (of Section 4.2), which is based on iterative probabilistic model checking. Although this measure is capable of generating the foreseen response time distributions, its underlying approach has two major drawbacks: (i) a simplified model is used that takes much human effort to create and yields approximate results; and (ii) evaluating a model takes (too) much time due to many and exhaustive iterations.

In this chapter, we take a threefold approach to eliminating both drawbacks, as follows. First, a literature study is conducted (in Section 7.2) which addresses four performance evaluation techniques followed by a number of performance toolsets (see also Appendix B).

Second, we conduct a case study on biplane interventional X-ray (iXR, [158, 159]) systems (in Section 7.3 and 7.4) in which we reuse the model calibration

mechanism (of Section 5.2.1 and 5.2.2) to inject measurements into the model. This case study puts emphasis on the above mentioned drawbacks, as follows. First, it brings about a complex model that has both nondeterminism, e.g., via parallel composition, and probabilism, e.g., due to model calibration. Second, evaluation may consist of many, computationally exhaustive iterations. This is the result of a wide range of possible latency outcomes in the case study.

Third, we built on existing work (of Chapter 6) to mitigate the two drawbacks, as follows. First, we extend iDSL with a way to automatically find a suitable simplified model, i.e., one that realizes an acceptable trade-off between accurate and fast results (in Section 7.5.1). Second, to save evaluation time, the evaluation starts with a sequential application of lightweight evaluation techniques, viz., basic estimates, simulations, traditional model checking (in Section 7.5.3 and 7.5.4), before probabilistic model checking.

This leads to a performance approach that fulfills three properties: (i) it explicitly yields latency distributions; (ii) it does not rely on statistics to generalize observations; and (iii) it provides a certain extend of modeling freedom.

.

7.2 Literature

In this section, we discuss four techniques to evaluate the system performance (cf. Section 2.2), followed by a number of performance evaluation toolsets (cf. Appendix B) that employ these techniques, as follows.

Techniques to evaluate system performance come in many flavors. *Basic estimations* employ widely used and generic spreadsheets or other high-level models [3] and lead to extremely fast but often inaccurate results, viz., they are not particularly suited to capture the system dynamics that parallel processing and scheduling bring about [13, 14].

Simulations [90, 133] provide relatively fast results by randomly exploring the system behavior via Monte Carlo sampling [143]. Statistics are used to generalize observations, which inherently leads to errors, especially in case of rare events [176, 180].

Analytic queuing methods [92], however, provide quick and accurate results. In return, they often require the distributions that are used to calibrate the model to be memoryless and can thus only be used for certain systems.

Exhaustive methods, e.g., (probabilistic) model checking [202, 203], potentially provide the required accuracy and flexibility in model choice [130, 203].

However, they do not scale well and typically require many computational resources and much time [203]. To make matters worse, they also suffer from the so-called state space explosion problem [36, 37].

Next, the literature study on performance evaluation toolsets (see also Appendix B) illustrates how the currently available performance evaluation toolsets fulfill the three properties (of Section 7.1).

Metropolis [8], the Hierarchical evaluation tool (HIT, [18, 199]) (cf. Appendix B.3) and MPA (Appendix B.7) separate the software and hardware model, as with the Y-chart philosophy [120], but do not specifically support response time distributions.

Platform Independent Petri net Editor (Appendix B.12) supports simulation and analysis, but also lacks explicit support for response time distributions.

The tagged customer approach [79] is a numerical method to compute response time distributions for Queueing Networks (QNs, [10, 70, 86, 92, 114, 196]). Software Performance Evaluation (of Appendix B.14) combines a software model of execution graphs, and machine model of QNs, as input for analysis. However, the tagged customer approach and SPE require models with memoryless distributions for efficient analysis.

The Palladio framework (Appendix B.9) bridges the gap between software and performance analysis via Unified Modeling Language [65] artifacts, whereas Software/Hardware Engineering (SHE, [59, 189]) provides a graphical user interface (GUI) to connect clusters and components. The Octopus toolset (Appendix B.8) is centered around an intermediate language. Both the Palladio framework, SHE and Octopus use simulations as their prime performance evaluation technique. They therefore rely on statistics to generalize their observations.

We conclude that none of the approaches above fulfill all the three properties (of Section 7.1).

7.3 The high-level performance model

To illustrate our approach, we evaluate the performance of *biplane* iXR systems, which generate 3D images based on two perpendicular planes, named frontal and lateral plane, of X-ray beams (see Figure 3.3(b)). A biplane iXR system (as depicted in Figure 3.1) consists of a table, arc and display. During surgery, the patient lies on the table, while the surgeon stands next to it. X-ray beams are

sent between both ends of the arc to record what is happening inside the body of the patient. The result is shown in high quality on the display after a small latency caused by Image Processing (IP). The latency of IP needs to continuously be below a certain threshold for proper hand/eye coordination [112], i.e., the surgeon perceives the images to be in real-time.

In the remainder of this section, we use the iDSL language (of Chapter 4) to specify *biplane* iXR systems. For this purpose, the six concepts of the iDSL language (as explained in Section 4.2) are represented, as follows. Section 7.3.1 contains the Process, Section 7.3.2 the Resource, Section 7.3.3 the System, Section 7.3.4 the Scenario, Section 7.3.5 the Measure, and Section 7.3.6 the Study.

7.3.1 The high-level Process

An iDSL process decomposes service requests into atomic processes. In this study, a biplane iXR system contains two processes that are based on the same process model (see Table 7.1).

The processes turn X-ray beams into high quality images via a pipeline that decomposes in processes “Noise_reduction" and “Refinement", at its highest level. In turn, process “Noise_reduction" decomposes into a sequence of five atomic processes, while process “Refinement" entails a choice between one or two calls of atomic process “Refine". The amount depends on the number of monitors attached to the iXR system, which is either one or two. The number of monitors is left unspecified by modeling it as a nondeterministic choice. Execution times of an atomic process are estimated by applying the Empirical Distribution Function (EDF, [5]) to a sample of 50 measured execution times; each measurement receives a weight of $\frac{1}{50}$.

7.3.2 The high-level Resource

A resource is capable of performing one atomic task at a time. In this study, biplane iXR systems are equipped with one CPU (see Table 7.2) to perform the individual steps of both processes.

7.3.3 The high-level System

A biplane iXR system comprises two similar services (see Table 7.3) that instantiate the same process model (of Section 7.3.1). For both services, all atomic processes, indicated by the “all” keyword, are assigned to resource CPU in a first-in first-out (FIFO) and non-preemptive manner.

Table 7.1: The code of an iDSL process with abstract loads

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

Table 7.2: The code of an iDSL resource

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

Table 7.3: The code of an iDSL system comprising two services

```
Section System
    Service Frontal_Image_Processing_Service
        Process Image_Processing
        Resource Image_PC
        Mapping assign { (all,CPU) }
          scheduling policy { (CPU, FIFO) }
    Service Lateral_Image_Processing_Service
        Process Image_Processing
        Resource Image_PC
        Mapping assign { (all,CPU) }
          scheduling policy { (CPU, FIFO) }
```

7.3.4 The high-level Scenario

In the study, biplane iXR systems process images at a fixed frame rate, for both services (of Section 7.3.3), viz., we consider images with fixed inter-arrival times of 40000 μs, for both services. On top of that, to study the effect of concurrency between the two IP chains, service lateral IP executes after a given

Table 7.4: The code of an iDSL scenario with two concurrent services

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

Table 7.5: The code of an iDSL measure with only one measure

```
Section Measure
  Measure CDF of ServiceResponse times
    via advanced PTA model checking
```

offset. This offset varies between different design instances by making it depend on the *offset* dimension in the design space (see Section 7.3.6). Clearly, an offset of 0 maximizes concurrency, while an offset of 20000 minimizes it.

7.3.5 The high-level Measure

In iDSL, measures of interest direct which metrics are to be retrieved and how. In this chapter, we use advanced model checking, which is the main contribution of this chapter, to return latency distributions (see Table 7.5). This measure is deterministic, i.e., multiple evaluation iterations all yield the same results.

7.3.6 The high-level Study

Finally, we define the study in Table 7.6, which has a design space that comprises a dimension named “offset" with possible values “0", “10000", “20000" and “30000". These values correspond to different degrees of concurrency (see Section 7.3.4). In the iDSL model, this offset is only used for the iDSL scenario (see Table 7.4). Namely, it is used to vary the offset time between service instances of both services. Hence, it only affects the degree of concurrency.

Table 7.6: The code of an iDSL study

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

7.4 The underlying performance model

In this section, we present the Modest code that the transformation of the iDSL model (of Section 7.3) brings about. We use therefore the six sections of the iDSL language (cf. Section 4.2): the Process (in Section 7.4.1), Resource (Section 7.4.2), System (Section 7.4.3), Scenario (Section 7.4.4), Measure (Section 7.4.5), and Study (Section 7.4.6).

7.4.1 The underlying Process

Section 7.3.1 shows the iDSL process of biplane iXR systems (see Table 7.1), which decomposes in a sequence of processes followed by either one or two times "Refine" (an alternative choice). This structure is directly translated to Modest processes similar to the Modest process in Chapter 4 (cf. Table 4.7).

7.4.2 The underlying Resource

Section 7.3.2 presents the iDSL resource of biplane iXR systems (see Table 7.2) containing a high-level resource "Image_PC" consisting of a resource "CPU" with a buffer. The resulting Modest code is similar to the functions "machine_call_..." and "machine_CPU" of Table 5.13, along with a buffer definition (cf. Section 5.14)

7.4.3 The underlying System

In Section 7.3.3, the iDSL system resource of biplane iXR systems is represented (see Table 7.3). It comprises two similar services of which all atomic processes are assigned to resource CPU in a FIFO way. In Modest, this mapping is realized by a function for each atomic process (of Section 7.1), which indirectly calls the CPU, similar to Table 5.15.

7.4.4 The underlying Scenario

In Section 7.3.4, the iDSL scenario contains two streams of requests (see Table 7.4), viz., for frontal and lateral, that have an offset between that depends on a design variable in the study (see Table 7.6). In Modest, each stream leads to two functions, similar to Table 4.10 but with different initial offsets and inter-arrival times. An init function that takes care of the initial offset, and a periodic function that repeats calling the respective service forever with a fixed period.

7.4.5 The underlying Measure

Section 7.3.5 contains a single measure (see Table 7.5), viz., one for computing service response time distributions using advanced model checking. Its functioning is explained elaborately in Section 7.5. The measure is referred to as advanced, because it is more developed than the previously introduced measure based on model checking (of Chapter 6 and Section 6.2.6).

7.4.6 The underlying Study

In Section 7.3.6, a 1-dimensional design space comprising four design instances was defined (see Table 7.5). Consequently, iDSL performs the complete evaluation (in Section 7.5) distinctly for four different “offset” values, viz., “0", “10000”, “20000” and “30000”. In effect, the evaluation gets performed for four levels of concurrency.

7.5 Performance model evaluation

In this chapter, we introduced a new measure “advanced model checking” for obtaining response time distributions for services, as referred to in Section 7.3.5 and 7.4.5). The measure builds on the measure of the previous chapter (of Section 6.2.7 and 6.3.5).

In this section, the functioning of the new measure is explained in the following four steps. In Section 7.5.1, we show how automatic model simplifications are iteratively applied to find the model with the right complexity. In Section 7.5.2, we present four performance evaluation techniques (see also Section 7.2), which are seamlessly combined into a sequential toolchain in Section 7.5.3. Finally, Section 7.5.4 provides the dataflow of this toolchain, which

concerns the way the inputs and outputs of the performance evaluation techniques are connected.

7.5.1 Automated model simplifications in iDSL

The model of a biplane iXR system (of Section 7.3) is hard to analyze for exact results, because of applying model calibration in the iDSL process (cf. Section 7.3.1). Despite this complexity, we would like to obtain exact response time distributions of this model using probabilistic model checking, which is known to be an evaluation technique that is prone to the state space explosion problem [36, 37].

Therefore, we present an automated model simplification toolchain (as depicted in Figure 7.1) that reduces complexity by simplifying the iDSL process, comprising three steps: (i) apply multiple combinations of two model simplification techniques (i.e., the clustering of measurements and changing the model time unit) to the iDSL instance, leading to an array of models, (ii) measure for

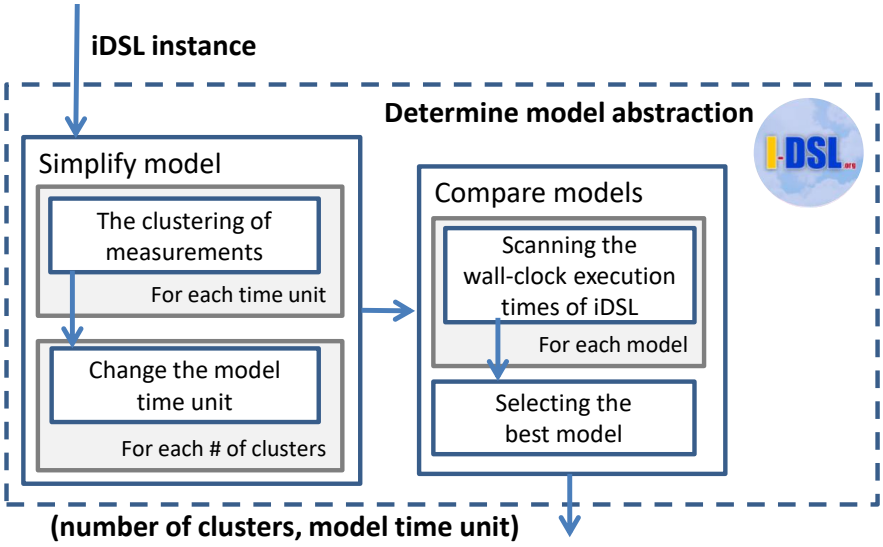
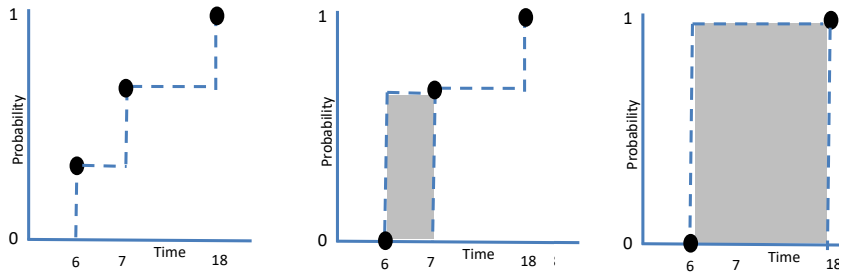


Figure 7.1: The iDSL tool chain for determining the model abstraction, which constitutes the tool chain for advanced model checking (in Figure 7.4).

each of these models the wall-clock execution time of iDSL for one probabilistic model checking iteration, and (iii) select the model that realizes the best trade-off between model complexity and execution time of analysis. In the following, we discuss these individual steps.

The clustering of measurements The first simplification method is applied to the EDF functions of atomic processes, which are based on a number of measurements. Measurements are clustered into a given number of clusters, using K-means clustering [115], which has the objective to cluster similar measurements together. For each cluster, its measurements are summarized by an interval of nondeterministic time, while all these intervals are combined via a probabilistic choice. Clustering reduces the model complexity, because it reduces the number of alternatives for selecting the execution times in the model.

In Figure 7.2, we show a small example based on three measurement values 6, 7 and 18. In Figure 7.2(a), we show the original EDF, which assigns an equal weight of $\frac{1}{3}$ to each of the 3 measurements. When the number of given clusters is greater than or equal to the number of measurements, this original EDF is returned since each measurement is assigned to its individual cluster. In Figure 7.2(b), the result of K-means clustering with 2 clusters is shown, viz., measurements 6 and 7 are grouped in one cluster due to their proximity, and 18 in the other. Consequently, 6 and 7 are represented by a nondeterministic time interval, which is graphically depicted as a grey area that covers time range $[6 : 7]$, and probability range $[0 : \frac{2}{3}]$. This grey area represents an ambiguity, namely all distributions that go through this area are possible ones. Finally, in



(a) 3 clusters (b) 2 clusters (c) 1 cluster

Figure 7.2: EDF based on measurements $6\mu s$, $7\mu s$ and $18\mu s$.

the case we indicate we only want one cluster, Figure 7.2(c) shows that all measurements are merged into this single cluster. This leads to a nondeterministic time range [6 : 18] and probability range [0 : 1].

The shown ambiguity introduced by the clustering of measurements implies a loss of information (loi). This *loss of information* per atomic process, inspired by the objective of K-means clustering [115], can be quantified as follows:

$$loi(A) = \sqrt{\frac{1}{k} \sum_{i=1}^{k} \sum_{\mathbf{x} \in A} (\mathbf{x} - \boldsymbol{\mu}_i)^2}, \tag{7.1}$$

where A is an atomic process represented by a set of measurements, k is the number of clusters, $\mathbf{x}$ a measured time, and $\boldsymbol{\mu}_i$ the average time of the measurements in cluster i. This measure considers for each measurement the distance to its cluster prototype, viz., the arithmetic mean of the measurements in the cluster. This rewards the clustering of similar measurements. Additionally, the loss of information of the whole process model P is then defined, as

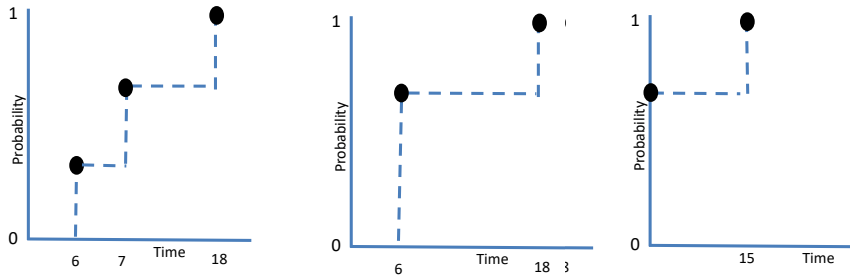
$$loi(P) = \frac{1}{|P|} \sum_{A \in P} loi(A). \tag{7.2}$$

where $|P|$ is the number of atomic processes P contains.

In iDSL, the clustering of measurements is implemented as an iDSL model transformation in which eCDFs (as introduced in Section 5.2.1) are replaced by probabilistic alternatives that are each nondeterministic ranges. To support the evaluation for different numbers of clusters, the number of clusters is added to the design space as a dimension that is hidden from the system designer.

Changing the model time unit The second simplification method increases the global time unit of the iDSL model. It is again applied to the EDF functions of each atomic process, viz., (i) the measurements are divided by the chosen time unit rounded to the nearest integer value; (ii) performance evaluation is applied; and (iii) the results of performance evaluation are multiplied by the chosen time unit again. This reduces complexity since the resulting model has less but bigger time steps, but also reduces precision, because of the rounding of measurement values.

In Figure 7.3, we show an example that is again based on measurements 6, 7 and 18. Figure 7.3(a), the case of time unit=1 is shown, which exactly matches



(a) time unit=1 (b) time unit=6 (c) time unit=15

Figure 7.3: EDF based on measurements $6\mu s$, $7\mu s$ and $18\mu s$.

the original EDF, viz., dividing measurements by 1 does not lead to rounding errors. In Figure 7.3(b), the case for time unit=6 is shown. Measurements 6 and 18 are not affected because they are multiples of 6, but measurement 7 induces a rounding error, viz., an integer division of 7 by 6 followed by a multiplication by 6 yields 6 instead of 7. Effectively, measurement 7 is replaced by 6 in the resulting graph, yielding two 6 and one 18 values. In Figure 7.3(c), we use time unit=15. Measurements 6 and 7 are both rounded to 0, whereas measurement 18 transforms into 15.

The *loss of precision* (lop) for each measurement x is then:

$$lop(x) = |x - \left\lfloor \frac{x}{t} \right\rceil \cdot t\,|, \tag{7.3}$$

where $\left\lfloor \frac{x}{t} \right\rceil$ is the nearest integer to $\frac{x}{t}$, t the model time unit.
Note that $lop(x)$ ranges from $lop(x) = 0$, for $t = 1$, to $lop(x) = x$ for $t \to \infty$. The loss of precision of a process model P is (similar to Equation 7.2):

$$lop(P) = \frac{1}{|P|} \sum_{A \in P} \frac{1}{|A|} \sum_{x \in A} lop(x), \tag{7.4}$$

where A is an atomic process model, P a process model, $|A|$ the number of measurements A contains, and $|P|$ the number of atomic processes P contains.

Note that Equation 7.2 and 7.4 are normalized using factors $\frac{1}{|P|}$ and $\frac{1}{|A|}$, which makes these measures universal. Namely, they can be used to compare the *loi* and *lop* of iDSL models with different structures, because they take the number

of atomic processes and number of measurements they contain, into account. However, this way of normalizing does not take the full structure of a process model into account. For instance, it neglects whether a process is of kind *seq*, *alt*, *palt*, etc. It should therefore be used with caution when comparing the loss of precision and information of processes with very different structures.

In iDSL, the change of model unit is implemented as a model transformation. Namely, all occurrences of time in the iDSL model, e.g, the execution times in the iDSL process and the arrival times in the iDSL scenario, are replaced by ones divided by the time unit. The actual evaluation of this quotient happens at Modest-level, since the occurrences of time may be design dependent. To support the evaluation for different model time units, the model time unit is added to the design space as a dimension that is hidden from the system designer.

Scanning the execution times of iDSL In our case study, we combine the two above model simplifications as shown, sequentially in the order given, to define a set of simplified Modest models. Let $\mathcal{M}_{n,t}$ be the simplified Modest model with n cluster segments and time unit t. We then define the following set of 11×11 models: $\{M_{n,t} \mid n,t \in \{1,2,\ldots,1024\}\}$ comprising 121 models. Note that the array size 11 and multiplication factor 2 for each dimension are variables of the approach that are adjustable in iDSL.

After this, iDSL performs one execution of probabilistic model checking for all these models to determine the amount of wall-clock needed per execution.

Table 7.7: Execution times (in seconds) of one probabilistic model checking call in iDSL for different model time units and number of clusters, for service Frontal IP, offset=20000.

	$\mathbf{t_{16}}$	$\mathbf{t_{32}}$	$\mathbf{t_{64}}$	$\mathbf{t_{128}}$	$\mathbf{t_{256}}$	$\mathbf{t_{512}}$	$\mathbf{t_{1024}}$	**loi**
$\mathbf{n_1}$	>99	40	7	5	3	3	3	.59
$\mathbf{n_2}$				>99	6	3	5	.44
$\mathbf{n_4}$					6	4	6	.33
$\mathbf{n_8}$					10	3	16	.22
$\mathbf{n_{16}}$					7	4	9	.15
$\mathbf{n_{32}}$					7	3	6	.07
$\mathbf{n_{64}}$					8	4	6	0
lop	8.3	17.5	37.1	80.0	180	377	395	

Table 7.7 shows the execution times of iDSL in seconds for service Frontal IP and offset=20000. The executions take place in the following sequential order: $((\mathcal{M}_{2^n,2^t})_{t=10}^{1})_{n=1}^{10} = [\mathcal{M}_{1,1024}, \mathcal{M}_{1,512}, \ldots, \mathcal{M}_{1,1}, \mathcal{M}_{2,1024}, \mathcal{M}_{2,512}, \ldots]$. This corresponds to starting in the top-right corner at $\mathcal{M}_{1,1024}$ in the Table 7.7, and going to the left step by step. Next, this is repeated for each line below (i.e., with $n = 2$, $n = 4$, ...) until $n = 1024$.

Since these execution times can be large, iDSL comes with a stop criterion to reduce the overall execution time of this "scan". For the time dimension, iDSL terminates the current execution and skips all remaining executions on the left, when the current execution exceeds a carefully selected time threshold, i.e., 99 seconds in our example. Table 7.7 also shows that models $\mathcal{M}_{1,16}$ and $\mathcal{M}_{2,128}$ exceed this threshold and thus all the models on the left of them have not been evaluated, since they are likely to take even longer. On top of that, also models positioned on the left and/or bottom of them have been skipped. Namely, they have more clusters and/or a smaller time unit, which makes them more complex.

Additionally, the loss of information is used for the clustering dimension. When it reaches 0, i.e., for $n = 64$ in the case study in which 50 measurements are used, no clustering takes place since each measurement has its own privately assigned cluster. In this case, increasing the number of clusters, e.g., to $n = 128$ in the case study, inevitably leads to a model that has already been scanned.

Selecting the best model When the execution times for different models are known, e.g., Table 7.7, iDSL is able automatically select a simplified model depending on which criteria the system designer prefers. iDSL takes (a combination of) the following five criteria into account: (i) the execution time of one call; (ii) the model time unit; (iii) the number of clusters; (iv) the loss of information; and, (v) the loss of precision. Note that a higher number of clusters is preferable, whereas for the other criteria lower values are preferred.

To optimally illustrate the effect of combining both model simplifications, we manually select model $\mathcal{M}_{256,4}$ here, since it ensures that both model simplification techniques are applied to the model. Furthermore, it executes fast enough to yield results in a reasonable amount of time, viz., 6 seconds per call.

7.5.2 Applying performance evaluation techniques to iDSL

In this section, we present four performance evaluation techniques that we have implemented in iDSL. They are basic estimates, average behavior, abso-

lute bounds and latency distributions (cf. Section). We explain them in the following, and combine them in the overall tool chain (cf. Section 7.5.3) and corresponding dataflow (cf. Section 7.5.4).

Basic estimates Basic estimates are very fast numerical computations that return an optimistic (but possibly inaccurate) bound of either the minimum or maximum latency in a way similar to asymptotic bounds in QNs. The result is optimistic because the concurrency between services and processing steps for resources are not taken into account. Basic estimates operate on a iDSL service process. They perform a recursive algorithm on its structure and return a latency value.

The best-case function traverses the whole service process recursively, distinguishing the following four cases: (i) for atomic processes, the taskload is returned; (ii) for (probabilistic) alternative processes, the minimum of all recursively evaluated children processes is returned; (iii) for parallel processes, the maximum of the evaluated children processes is returned; and, (iv) for sequential processes, the sum of the evaluated children is returned. The best-case function returns a theoretical minimum latency; the real minimum latency is never lower than this latency.

The worst-case function operates similarly, but returns the maximum while evaluating (probabilistic) alternative processes.

Average behavior Average behavior is observed via simulation runs that each return a sequence of latencies. We use 4 runs with an as soon as possible (ASAP) scheduling policy, and 4 with an as late as possible (ALAP) policy [87] to maximize variation of the results. The minimum simulated result is an upper bound for the lower bound, and vice versa. We compute this using the MODES simulator [83] on a Modest model, automatically derived from an iDSL model (as in [202–206] and Chapter 6).

Absolute bounds Absolute bounds mark the absolute minimum and maximum latency possible. They are a refinement of basic estimates, e.g., they do take concurrency into account. They can be obtained via model checking in which probabilities are neglected, i.e., the probabilistic choices in the model are replaced by nondeterministic equivalents. As a result, it is not possible to ascertain the probability at which these bounds occur.

We compute an absolute bound by performing a binary search on a given range marked by a minimum and maximum value (see also Section 4.4.4 and

Table 4.13). We iteratively apply the MCSTA model checker [83] to a Modest model that is derived from an iDSL model (as in [202–206] and Chapter 6).

Latency distributions Latency distributions show, for each time, the probability that the latency is less than or equal to a certain value. They are exhaustively obtained via iterative probabilistic model checking (cf. Section6.4.4), i.e., each iteration yields a probability for a given latency. Hence, the MCSTA model checker [83] is iteratively applied to a Modest model that is derived from an iDSL model (as in [203, 205] and Chapter 6).

7.5.3 The advanced model checking toolchain

In this section, the automated model simplifications (of Section 7.5.1) and performance evaluation techniques (of Section 7.5.2) are combined to form a toolchain (as depicted in Figure 7.4) that efficiently returns service latency distributions. The sequential toolchain is executed, for each design instance and service, using the following six steps:

1. The model abstraction is determined. That is, the wall-clock iDSL execution times of models of different abstractions are determined, after which the best model is selected. The selected model is used throughout the remainder of the tool chain.
2. Basic estimates are quickly computed on the basis of iDSL processes. This yields an optimistic best and worst case, i.e., concurrency is neglected.
3. Multiple simulation runs are performed to get an impression of average behavior. We use 4 runs using ASAP scheduling, and another 4 using ALAP scheduling each, consisting of 50 service requests. This is a trade-off between time spent on simulation, and model checking (in Step 4).
4. Model checking is performed to compute the absolute minimum and maximum latencies. iDSL models may contain nondeterminism whose resolution affects the outcomes of the latencies. Hence, we introduce minimum and maximum time (t_{min} and t_{max}), for all resolutions of nondeterminism. This yields four model checking computations, viz., either t_{min} or t_{max}, and either the minimum or maximum latency. The corresponding four latency ranges depend on basic estimates and simulations (as explained in Section 7.5.4).

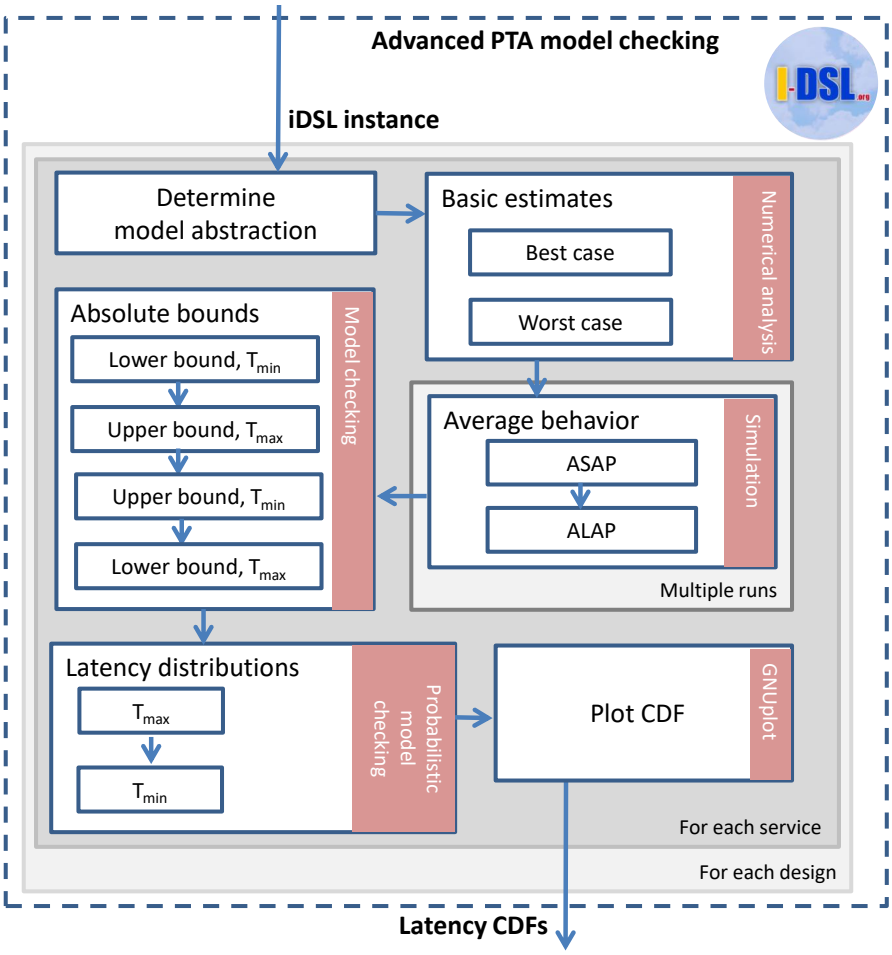


Figure 7.4: The iDSL tool chain for advanced model checking. “Determine model abstraction" is decomposed in Figure 7.1.

5. The latency distribution functions are computed within ranges (of Step 4), using the algorithm of Section 7.5.2. Intermediate results are plotted in real-time to enable a so-called anytime algorithm (see Section 7.6.3).
6. The computed probabilities are plotted to a graphs via GNUplot.

7.5.4 Dataflow between performance evaluation techniques

In Section 7.5.3, we provided the six steps of the toolchain that execute in sequence. They are designed to perform in the given order due to dependencies that exist between them. On top of that, the dataflow of the toolchain (of Figure 7.5) illustrates for the components “Basic estimates", “Average behavior" and “Absolute bounds", how their inputs and outputs connect.

The absolute bounds comprise four functions that all perform binary searches on an input range $[min : max]$. When this input range is wide, many expensive iterations of model checking are required to find the final result. However, when this range is too narrow, the final result might not be in the range and, hence, impossible to find. Ideally, the input range is both narrow and contains the result. In the following, we use the outcomes of the basic estimates and average behavior to yield ranges that are both narrow and valid.

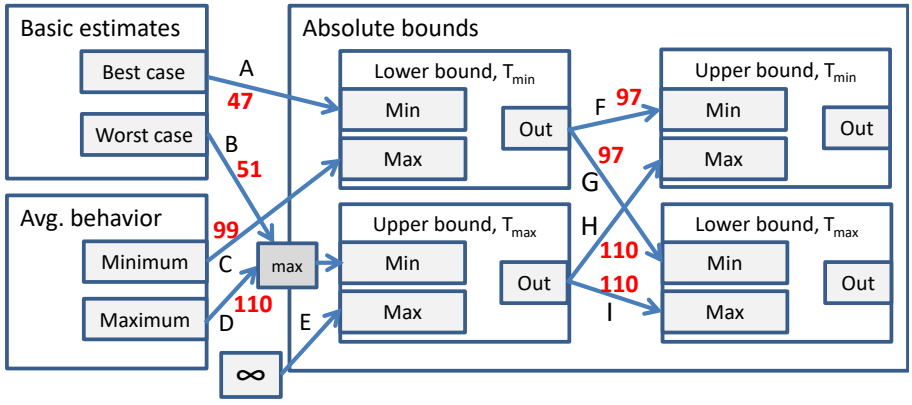


Figure 7.5: The dataflow of “Basic estimates", “Average behavior" and “Absolute bounds". On the edges, execution times for service Frontal IP and offset=0.

In Figure 7.5, the best case (**A**) and worst case (**B**) form minima for the lower bound t_{min} and upper bound t_{max}, respectively. Namely, they output optimistic bounds and, hence, provide safe lower bounds for the respective ranges. Next, the minimum simulated result (**C**) serves as a maximum for the lower bound t_{min}. After all, a lower bound can never be greater than any simulated result by definition. Something similar holds for the maximum simulated result (**D**), which marks a minimum for the upper bound t_{max} and so does the worst case. Hence, we can take the maximum of both results (**B**+**D**) to make the range as small as possible.

A (tight) maximum for the upper bound t_{max} (**E**) is hard to find, because it depends on properties that can be hard to understand, such as the system dynamics that parallel processing and scheduling bring about [13, 14]. Hence, infinity is used for the upper bound. In practice, the binary search algorithm finds a finite upper bound by doubling and evaluating the lower bound repetitively, until a probability of 1 has been found, similar to the initial scan (of Section 6.4.4).

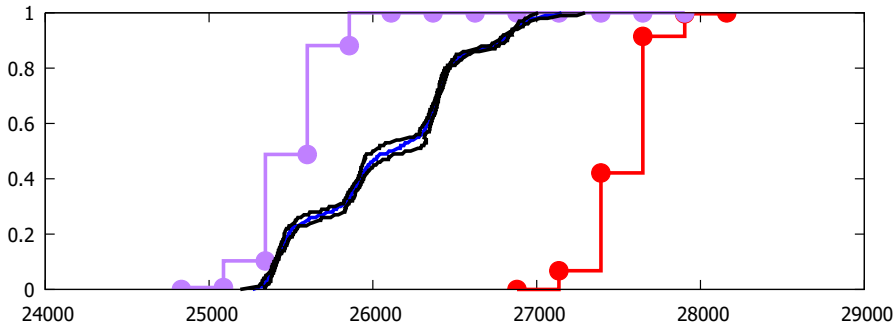
We conclude with the upper bound t_{min} and lower bound t_{max} that operate on the same range. Namely, they receive the output of lower bound t_{min} (**F**+**G**) as a minimum, and the output of upper bound t_{max} as a maximum (**H**+**I**). These connections are valid for two reasons: (i) a lower bound is always smaller than or equal to an upper bound of the same kind (i.e., t_{min} or t_{max}); and (ii) a value for t_{min} is always smaller than or equal to a value for t_{max}. Eventually, the four outputs of the absolute bounds form two input ranges for computing latency distributions using probabilistic model checking (not shown in the figure).

7.6 Performance results

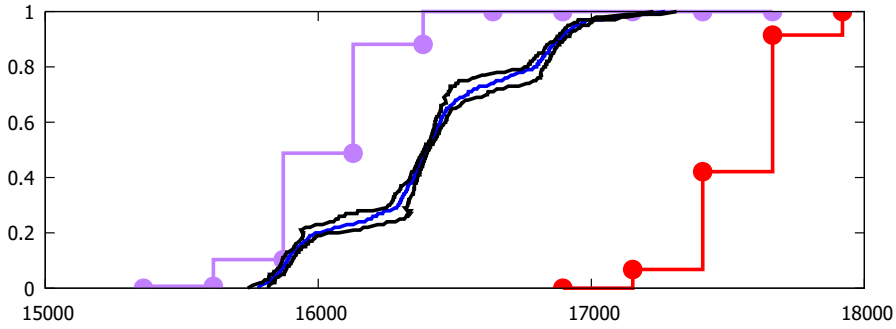
In this section, we check the validity of the case study results by comparing them with simulation outcomes and the basic estimates (in Section 7.6.1), assess the efficiency of the approach (in Section 7.6.2), and show how an anytime algorithm is useful (in Section 7.6.3).

7.6.1 The validity of the approach

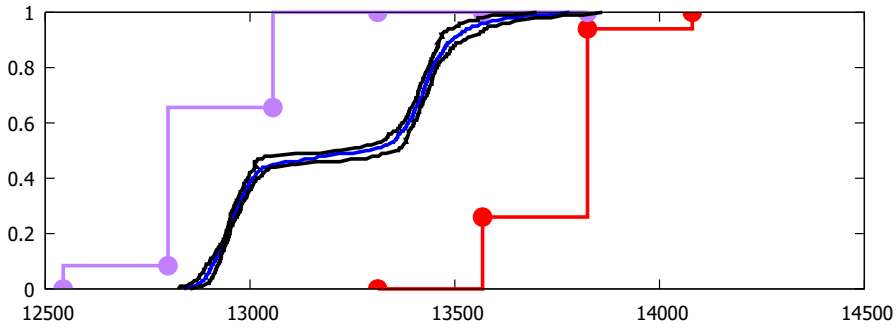
In the following, we check the validity of the case study results by comparing them with: (i) simulation outcomes; and, (ii) basic estimates. For this purpose, Figure 7.6 conveys latency distributions for service Frontal IP for four designs (Figure 7.6(a) till 7.6(d), respectively), generated by iDSL for the case study.



(a) service = Frontal IP, offset = 0

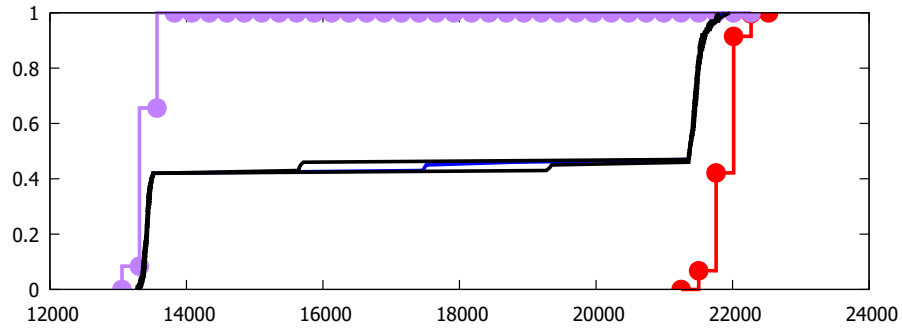


(b) service = Frontal IP, offset = 10000



(c) service = Frontal IP, offset = 20000

Figure 7.6: The lower (in purple) and upper bound CDF (in red), the simulation average (in blue), and $\alpha = 0.95$ Confidence interval (in black) for four designs.



(d) service = Frontal IP, offset = 30000

Figure 7.6: The lower (in purple) and upper bound CDF (in red), the simulation average (in blue), and $\alpha = 0.95$ Confidence interval (in black) for four designs.

Comparison with simulation We check whether the case study results are reasonable with respect to the model by comparing them with simulation outcomes on the same model. In Figure 7.6(a) till 7.6(d), the results of the case study for each design comprise a lower bound CDF (in purple) and upper bound CDF (in red). Additionally, each figure contains a simulation average CDF (in blue) and simulation Confidence Interval ($\alpha = 0.95$) CDFs (in black), which is based on 4 simulation runs of 200 requests.

The lower and upper bound are valid, when, for each probability, the lower bound is smaller then all simulation CDFs and the upper bound is greater than all simulation CDFs. In Figure 7.6(a) till Figure 7.6(d), this property is not violated, although the bounds and simulation results are sometimes close, e.g., for latency=15800 in Figure 7.6(b). It is also important to realize that when the lower bound and upper bound CDFs are close to each other, they provide more information (cf. Section 2.3.2). However, depending on the amount of nondeterminism in the model, they might have to be far apart to be valid for all conceivable simulation outcomes.

Comparison with basic estimates We verify whether the case study results are plausible by comparing them with the basic estimates, which are $12729\mu s$ (best case) and $14033\mu s$ (worst case), which do not taking concurrency into account.

In Figure 7.6(c) (offset=20000), the case in which no concurrency occurs

is shown. Theoretically, the lower bound CDF (in pink) needs to be above the the best case for all probabilities. The smallest lower bound for a probability greater than 0, which is 12800μs, is greater than 12729μs albeit slightly.

Again in Figure 7.6(c), the upper bound CDF (in red) needs to be above the worst case estimate for all probabilities. However, the smallest upper bound for a probability greater than 0, which is 13568μs, is smaller than the worst case value 14033μs. We attribute this to one of the model simplification techniques in which we have selected the model time unit to be 256 (cf Section 7.5.1), viz., this technique is prone to causing errors by rounding times.

Next, we consider the maximum concurrency case of Figure 7.6(a) (offset=0). It reveals, as expected, that the latencies are roughly and up to twice as high as the ones of the offset=20000 case (of Figure 7.6(c)), due to the fact that maximally two services are competing for one CPU resource.

Finally, Figure 7.6(b) and 7.6(d) (offset=10000 and 30000) are designs with a medium level of concurrency. As anticipated, they yield latencies that lie between the minimum and maximum concurrency cases (of Figure 7.6(c) and 7.6(a)).

7.6.2 The efficiency of the approach

In this chapter, previous work has been extended with two automated model simplification techniques, basic estimations and simulation, and model checking. They have added complexity to the performance evaluation approach and required efforts to create. In this section, we justify the increased complexity and made efforts by illustrating how they have made the approach more efficient, as follows.

Model simplification techniques Table 7.8(a) shows the wall-clock execution times of iDSL for different components and designs, for the iDSL model of this chapter (cf. Section 7.3). It can be seen that the automated model simplification techniques (column *ms*) consume a significant amount (ranging from 31% to 69%) of the total execution time (column *sum*) of iDSL, while in Chapter 6 no time was spend on this activity. However, the alternative would be a manual search for the right simplified model (as in Section 6.2.2), which is not only labor intensive but also error prone.

Basic estimates and simulations In the following, we make clear how basic estimates and simulations make the approach more efficient. To illustrate this,

Table 7.8: The iDSL performance for four components, viz., model simplification (ms), simulation (sim), model checking (mc), and prob. model checking (pmc).

(a) Execution times (in seconds) on a PC (Intel i7-2670QM 2.20GHz, 24Gb RAM).

	Frontal IP service				
offset	**ms**	**sim**	**mc**	**pmc**	**sum**
0	499 (31%)	27 (2%)	758 (48%)	305 (19%)	1589 (100%)
10000	646 (57%)	27 (2%)	333 (29%)	135 (12%)	1141 (100%)
20000	450 (69%)	26 (4%)	134 (20%)	46 (7%)	656 (100%)
30000	619 (48%)	25 (2%)	313 (24%)	324 (25%)	1281 (100%)

(b) Number of probabilistic model checking calls.

	Frontal IP			Lateral IP			
offset	**ms**	**mc**	**pmc**	**ms**	**mc**	**pmc**	**sum**
0	16	27	7	16	27	6	99
10000	26	18	7	26	21	8	106
20000	26	13	4	26	13	4	86
30000	26	21	8	26	19	19	119

we have augmented the dataflow in Figure 7.5 with latency numbers for service Frontal IP and offset=0 (in red). The input ranges of the absolute bounds, which depend on the outcomes of basic estimates and simulations, are $[47 : 99]$, $[110 : \infty]$, $[97 : 110]$ and $[97 : 110]$. We consider them fairly tight, since they would each be $[0 : \infty]$ without basic estimates and simulations. On top of that, the execution times of basic estimates and simulations are negligible, viz., only up to 4% of the total execution time, as indicated in Table 7.8(a).

In concrete, less model checking iterations are needed, which we demonstrate by comparing two executions of the iDSL chain, viz., one without and one with using basic estimates and simulations, for service “Frontal_IP" and offset=0. The total wall-clock execution time without basic estimates and simulations is 1937 seconds for 61 calls (not shown in Table 7.8(a)), opposed to only 1589 seconds for 50 calls with basic estimates and simulation included (in Table 7.8(a)).

Model Checking We convey that replacing probabilistic model checking iterations with model checking ones, wherever possible, makes the approach faster.

To this end, we test whether model checking iterations require, on average, less wall-clock execution time than probabilistic model checking ones. For illustration, we consider the case for which offset=0 and Frontal IP service.

The average execution time for probabilistic model checking is the quotient of the execution time (Table 7.8(a), column "pmc"), which is 305 seconds, and the number of calls (Table 7.8(b), column "pmc" of Frontal IP). which is 7. The average execution time for model checking is the quotient of the execution time (Table 7.8(a), column "mc"), which is 758 seconds, and the number of calls (Table 7.8(b), column "mc" of Frontal IP). which is 27.

Conclusively, 7 probabilistic model checking iterations take 305 seconds, which is on average 44 (305/7) seconds each, whereas 27 model checking iterations take 758 seconds. This is on average (758/27) only 28 seconds each, a 27% reduction.

7.6.3 An anytime algorithm enabling intermediate results

In general, computations based on probabilistic model checking take significantly longer to execute than simulation-based ones, as demonstrated in both the previous chapter (see Table 6.11) and the current chapter (see Table 7.8(a)). Therefore, it might be interesting for the system designer to be able to obtain intermediate results as the evaluation goes. For this purpose, iDSL has been equipped with a so-called *anytime algorithm* [182] (as illustrated below), which is an algorithm that can always return results, even if it is interrupted before ending. However, the algorithm does need more runtime to obtain better results. Therefore, to appreciate the valuable time of the system designer, the objective of the iDSL anytime algorithm is to deliver intermediate results that are as useful as possible, as soon as possible.

For illustration, we show six figures with intermediate results for service Frontal IP, offset=0, and the lower bound (in Figure 7.7) that iDSL has automatically generated in one go with performance evaluation. It can be seen that

- most of the six figures with intermediate results are very similar to the final graph (especially Figure 7.7(d), 7.7(e) and 7.7(f)); and that,
- each of these six figures with intermediate results is not an approximation but merely an incomplete result, i.e., the area between the lower and upper bound is larger for earlier intermediate results

For these two reasons, the anytime algorithm allows the system designer to reduce the evaluation time, e.g., by terminating an ongoing computation in an early stage and settle for the current, incomplete result that might already answer his final question.

The key of the anytime algorithm is to select the next latency for which a probability is going to be computed, during probabilistic model checking. In iDSL, the algorithm considers differences in probabilities. That is, the algorithm computes the in-between latency for two neighboring latencies that differ most in probability values.

For illustration, we compare the third and fourth intermediate result (cf. Figure 7.7(c) and 7.7(d)), viz., we show what the anytime algorithm decides in Figure 7.7(c) and the outcome of that decision in Figure 7.7(d).

In Figure 7.7(c), the anytime algorithm has three options for selecting a latency to compute a probability for, viz., either 25088 (the middle of range [24832 : 25344]), 25472 (the middle of [25344 : 25600]), or 25856 (the middle of [25600 : 26122]). In this particular case, the anytime algorithm selects the third option (25856), because its probability difference of 0.51 (cf. Equation 7.7) is greater than 0.39 (cf. Equation 7.5) and 0.10 (cf. Equation 7.6). Consequently, the most recently computed probability 0.88 for latency 25856 has become visible in Figure 7.7(d) in constrast to Figure 7.7(c).

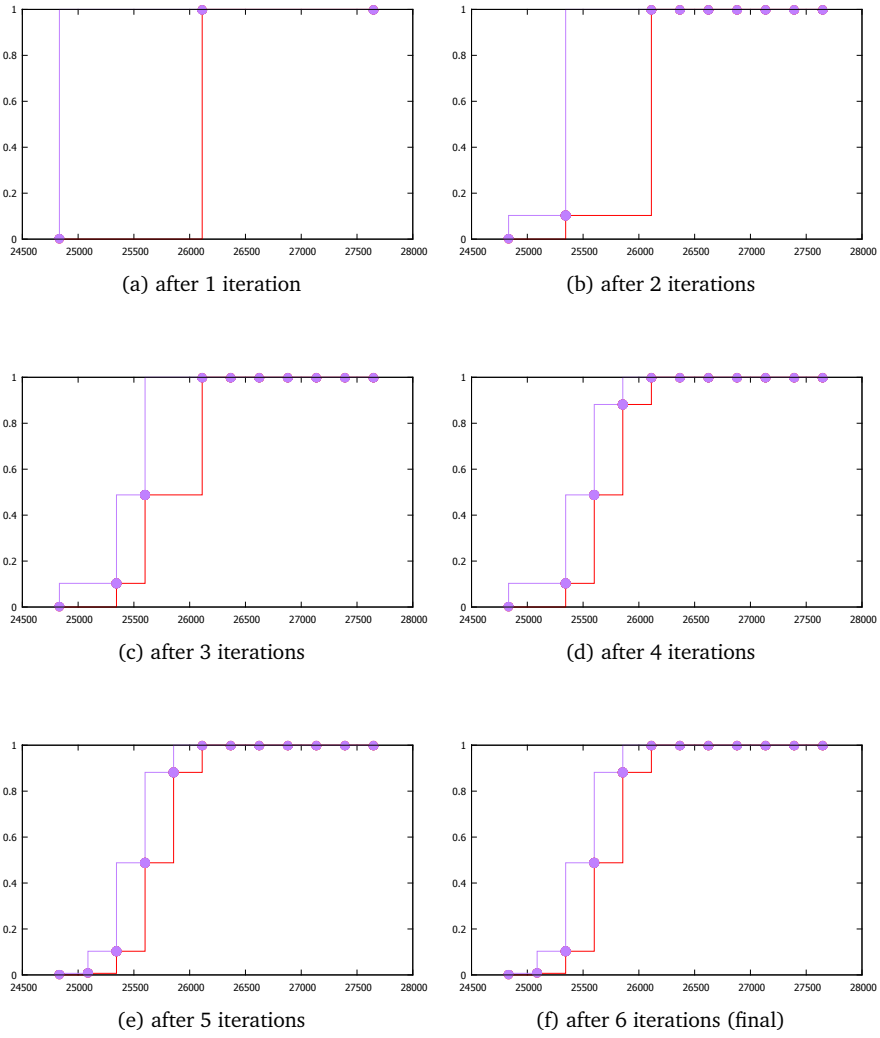
$$cdf(25088) - cdf(24832) = 0.10 - 0 = 0.10 \tag{7.5}$$

$$cdf(25600) - cdf(25088) = 0.49 - 0.10 = 0.39 \tag{7.6}$$

$$cdf(26122) - cdf(25600) = 1 - 0.49 = 0.51 \tag{7.7}$$

where *cdf* is the cumulative probability given a time (of Figure 7.7(c)).

For another illustration, we compare the fourth and fifth intermediate result (cf. Figure 7.7(d) and 7.7(e)) in which the probability of latency 25088 (the middle of range [24832 : 25344] is computed, while two other ranges, i.e., [25344 : 25600] and [25600 : 25856], display a greater probability difference. However, due to chosen the model time unit of 256 (see Section 7.5.1), only probabilities for latencies that are multiples of 256 can be computed. Hence, 25088 is the only latency for which the probability still needs to be computed.



(a) after 1 iteration

(b) after 2 iterations

(c) after 3 iterations

(d) after 4 iterations

(e) after 5 iterations

(f) after 6 iterations (final)

Figure 7.7: Anytime (maximum time) intermediate results after each iteration step, for service Frontal IP, offset = 0, the lower bound.

7.7 Conclusion

In this chapter, we have built on an exact measure to compute response time distributions (of Chapter 6) for service systems (of Section 4.2).

The new iDSL measure is again based on iterative probabilistic model checking, but it improves the measure of Chapter 6 on the following two aspects. First, it automatically applies model simplifications to the model, depending on a combination of the complexity of the model and the computer on which iDSL runs. For this purpose, iDSL benchmarks models of different complexities, i.e., it checks how long a probabilistic model checking iteration takes to run. This is in contrast to the measure of Chapter 6 that requires some manual configuration by the system designer.

Second, lightweight evaluation techniques are used at the start of the evaluation process that execute very quickly and yet provide a great deal of information. In return, less probabilistic model checking iterations need to be performed (see Table 7.5). Overall, the combination of evaluation techniques reduces the evaluation time and yet yields results that are exactly the same as the measure of Chapter 6, i.e., the use of lightweight techniques does not compromise the results.

Model validation is achieved by comparing the evaluated results with basic estimates that neglect concurrency. It conveys that the generated results are plausible with respect to what the basis estimates tell us.

CHAPTER 8

Evaluating load balancers for performance and energy-efficiency

This chapter is based on the following publication.

[204] F. van den Berg, B. Postema, and B.R. Haverkort. Evaluating load balancing policies for performance and energy-efficiency. In *Quantitative Aspects of Programming Languages and Systems*, volume 227 of *Electronic Proceedings in Theoretical Computer Science*, pages 98–117, 2016. doi: 10.4204/eptcs.227.7

8.1 Introduction

Recently, so-called *green* ICT solutions [123,125,146] in which saving electricity is an important goal, has gained a lot of attention. A major energy consumer in ICT are data centres, which consist of many servers that process some load that is distributed among various computing resources, e.g., computers, network links or processing units. A server environment is set up in a data centre, which provides the infrastructure that enables servers to function.

A reduction of energy consumption by servers can be achieved by making use of so-called dynamic power management. *Dynamic power management* [21, 138] allows switching between power states of servers to reduce power consumption, while trying to keep the performance intact, e.g., bringing to sleep underutilised servers. There are two key elements that constitute a power management *policy* [52, 155], namely: (i) *strategies* and (ii) *load balancing*. First, power management strategies specify when servers should switch

between the power states, e.g,. when a server is in sleep it will use less energy but require some initial startup time before being able to process an incoming load again. Second, the load should be balanced among the servers such that optimal performance is obtained. Generally, the load balancing is based on performance-related variables of the current system state, such as queue lengths.

Using this context, we would like to demonstrate that iDSL can be applied to a *domain* significantly different from interventional X-ray (iXR, [158, 159]) systems (of Chapter 3), viz., load balancers. We would also like to show that iDSL can be used to evaluate an other *metric* in addition to performance, viz., energy consumption. To realize this, we propose a combined performance and energy-consumption evaluation approach, using iDSL, that aims to find optimal power management policies. This approach should meet five objectives, i.e., it should

O1 (formally) model the performance and energy use of a load balancer;
O2 (formally) model a load balancer policy;
O3 be applicable to a (small) load balancer case;
O4 evaluate many differrent designs to find optimal policies regarding performance and energy consumption; and,
O5 yield performance and energy results that are valid.

In this chapter, we realize these objectives using the following five steps.

1. We extend iDSL with a load balancer construct in the process, as well as augment an iDSL resource with a notion of energy to obtain energy related results. Also, servers can be put to sleep when idling, with a simple timeout mechanism [21].
2. We equip the load balancer in iDSL with a powerful, yet concise, policy language, which covers strategies that observe, among others, the size of the queue to decide to which server jobs are assigned.
3. We construct a small load balancer example consisting of four servers. In this chapter, it shows that using four servers is sufficient to convey various load balancing patterns and moreover relatively easy to compute. The example is inspired by the so-called Peregrine cluster [154] at the Center for Information Technology (CIT, [35]). We assume that CIT decides to actively use dynamic power management features for their Peregrine cluster combined with load balancing. This cluster contains 4368 cores with four types of nodes, namely: (i) 162 nodes with 2×24 Intel Xeon 2.5 GHz

cores; (ii) 4 nodes with Nvidia K40 GPU accelerator cards; (iii) 2 nodes with Intel Xeon Phi 7120 accelerator cards; and (iv) 7 nodes with 4×48 Intel Xeon 2.6 GHz cores. A standard node consumes ca. 40 W for only the CPU cores [106]. To reduce complexity, the system is modeled using ten assumptions (see Table 8.1).

4. Pareto optimal [33] trade-offs between performance, i.e., the average service latency, and power, i.e., the average power (in Watt) by the four servers, are detected via simulations. We evaluate a large set of design instances that are represented as a composition of three variables, viz., (i) *queue size threshold*, (ii) *idle time-out*, and (iii) the way of *resolving nondeterminism*. Trade-off plots are generated by reusing existing iDSL functionality (see Section 5.5.1 and 5.5.2).
5. Model validation is accomplished by evaluating the performance and energy consumption for many designs using iDSL, and comparing the outcomes of a supposedly equivalent implementation in Anylogic [204]. This implementation is based on the framework for modeling and simulation of cloud computing infrastructures and services Cloudsim [30], and has been constructed at the same time.

8.2 The high-level performance model

The six concepts of the iDSL language (as carefully explained in Section 4.2) are explicitly represented, as follows. Section 8.2.1 contains the process, Section 8.2.2 the resource, Section 8.2.3 the system, Section 8.2.4 the scenario, Section 8.2.5 the measure, and finally Section 8.2.6 contains the study.

8.2.1 The high-level Process

The process of the load balancer (see Table 8.2) consists of a policy, a configuration, and four processes. It is instantiated using the process algebra construct *lbalt*, which has been added to iDSL to accommodate load balancing. It is named *lbalt*, because it falls in the category of the process algebra *palt* and *alt*-constructs as well as the iDSL-specific *desalt*-construct (see Table 5.12). That is, they all represent a selection of *exactly one* subprocess out of multiple ones.

Implementing an *lbalt* as a process algebra construct as just mentioned, provides a high degree of modeling freedom to the system designer. For instance, the system designer can interchangeably use the *palt*, *alt*, *lbalt* and

Table 8.1: Assumptions used to model a small load balancer example

$\mathcal{A}1$	Incoming requests arrive according to a Poisson process with negative exponentially distributed inter-arrival times with rate 1 request per second.
$\mathcal{A}2$	The system consist of four similar resources (or server nodes).
$\mathcal{A}3$	A load balancing policy specifies how incoming requests are distributed over the four servers.
$\mathcal{A}4$	A server has four power states; it is either switched on ("stateOn"), asleep ("stateSleep"), moving from "stateOn" to "stateSleep" ("stateSuspend"), or moving from "stateSleep" to "stateOn" ("stateWakeUp").
$\mathcal{A}5$	A server can only process tasks in state "stateOn".
$\mathcal{A}6$	A Server spends exactly 10 seconds in transition states "stateSuspend" and "stateWakeup" each, when changing states.
$\mathcal{A}7$	A servers consumes 14 Watt in state "stateSleep" and 200 Watt in other states, based on empirical studies [12, 68]
$\mathcal{A}8$	A server has an infinite queue with a nonpreemptive FIFO scheduling policy.
$\mathcal{A}9$	A server processes incoming requests deterministically with rate 1.
$\mathcal{A}10$	Evaluating a load balancing policy takes no (or negligible) time.

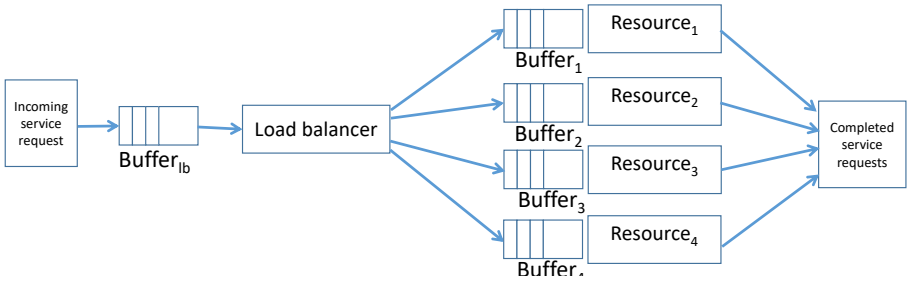


Figure 8.1: A load balancer distributes incoming requests from buffer$_{lb}$ over buffer$_n$ of resource$_n$, where $n \in \{1, 2, 3, 4\}$ is selected using a policy.

desalt-constructs during modeling, and simply replace one with another later in design. Also, an iDSL model with multiple *lbalt*-constructs enables the analysis of tandems, hierarchies, and other structures of load balancers.

In the following, we describe an *lbalt* construct by explaining it constituents, which are its policy, configuration, and subprocesses.

Lbalt policy A load balancer policy prescribes how a load balancer behaves with respect to distributing incoming requests to resources (as graphically depicted in Figure 8.1). Each time an incoming requests arrives it is placed in buffer_{lb}. For each incoming request, a load balancer has to select one of the resources to delegate this request to, which is done using the following algorithm: (i) system state variables relevant to load balancing are retrieved, e.g., the queue sizes of the resources; (ii) the preference of each individual resource is determined via a policy, which is an arithmetic expression that may include system state variables; (iii) the incoming request is delegated to the incoming buffer of the most desirable resource, which is the one with the lowest outcome for the arithmetic expression; and, (iv) when this resource finishes processing, it outputs the result to the completed service requests.

We illustrate what a policy entails by providing five example policies ($\mathcal{P}_0$ till $\mathcal{P}_4$), which we will combine to specify a more extensive policy $\mathcal{P}_5$ to be used in the remainder of this chapter. For more detail, the syntax (see Table 8.8), the concrete syntax of $\mathcal{P}_5$, and the translation of $\mathcal{P}_5$ to Modest can be found in Section 8.3.1. Let $\mathcal{P}_0$ be a policy that assigns incoming requests to the resource that currently has the shortest queue, i.e., the smaller the queue size, the lower the outcome of the arithmetic expression:

$$\mathcal{P}_0 = queueSize \tag{8.1}$$

where *queueSize* is the queue size of the resource under evaluation.

Next, let $\mathcal{P}_1$ be a policy that initially delegates all incoming requests to a fixed resource, but starts delegating requests to other resources when this resource is too busy, i.e., the smaller the queue size, the lower the outcome of the arithmetic expression, but turned-on resources are preferred to some extend:

$$\mathcal{P}_1 = queueSize + 20 \cdot (1 - stateOn), \tag{8.2}$$

where *stateOn* is the indicator function that returns 1, if and only if the evaluated resource is in state *On*, and 0 otherwise. To be precise, in the particular

case the queue size of the only turned-on resource exceeds 20, a second resource has to be turned on to compensate for the high load. In a similar fashion, a third resource has is turned on if the two resources have a queue size greater than 20, and so on. Note that the evaluation of policy P_1 only yields natural numbers.

Now, let $\mathcal{P}_2$ be a policy that yields a nondeterministic decision among the resources, i.e., it behaves like the process algebra *alt*-construct:

$$\mathcal{P}_2 = 0. \tag{8.3}$$

The policy always evaluates to 0, for all resources, and does not specify which resource gets selected, cf. nondeterministic choice. Contrarily, policy $\mathcal{P}_3$ yields a probabilistic choice, i.e., it behaves like the process algebra *palt*-construct:

$$\mathcal{P}_3 = random, \tag{8.4}$$

where *random* is a random generator that returns unique, real numbers uniformly distributed on range $[0 : 1]$. Hence, the resource with the lowest random number gets selected. This selection is uniformly distributed, because each resource has the same probability of receiving the lower number. Finally, policy $\mathcal{P}_4$ orders all resources and ensures a fixed, most preferred resource gets selected:

$$\mathcal{P}_4 = \frac{\text{ID}}{\text{numServers}}, \tag{8.5}$$

where $ID \in \{0, 1, \ldots, \text{numServers}\}$ is the unique identifier of the resource under evaluation, and numServers the total number of resources.

Below, we define policy $\mathcal{P}_5$ that we will use throughout the remainder of this chapter. Table 8.2 contains its implementation in iDSL. Policy $\mathcal{P}_5$ resembles policy $\mathcal{P}_1$, but has the number 20 replaced by a variable that is defined by design space dimension q (to be defined in Table 8.7). The resulting policies (for different values of q) do allow nondeterminism to occur, i.e., resources that have the exactly same queue size and power state, yield the same result for the arithmetic expression. In this case, the load balancer has to select a resource in a nondeterministic manner.

In a performance evaluation context, evaluating models with a high degree of nondeterminism can lead to variable results, i.e., each way of resolving nondeterminism can imply a wide array of performance outcomes. To counteract this, we eliminate this undesired nondeterminism by adding another selection

Table 8.2: An iDSL process for a load balancer

```
Section Process
    ProcessModel p seq {
      lbalt
        policy { ((queueSize+
                 ([dspace("q")]*([1]-stateOn)))
                 +[dspace("nd_res")]) }
        configuration { powerOn 200
                        powerShutdown 200
                        powerSleep 14
                        powerStartup 200
                        timeStartUp 10
                        timeShutdown 10
                        timeOutTime dspace("to") }
        { atom p1 load 1, atom p2 load 1,
          atom p3 load 1, atom p4 load 1 }
    }
```

mechanism to $\mathcal{P}_5$ that selects a resource in case policy $\mathcal{P}_5$ yields multiple resources. In concrete, this is accomplished by adding either policy P_3 (random selection) or policy P_4 (priority selection), as defined by the design space dimension *nd_res* to P_5. The result is a deterministic policy, because P_1 yields natural numbers and P_3 or P_4 to unique fractions $\in [0:1)$, as follows:

$$\mathcal{P}_5 = \begin{cases} queueSize + q \cdot (1 - stateOn) + random, & \text{if nd\_res=random,} \\ queueSize + q \cdot (1 - stateOn) + \frac{ID}{numServers}, & \text{otherwise.} \end{cases} \tag{8.6}$$

where q and *nd_res* are design dimensions of the iDSL study (see Table 8.7) that vary per design instance.

Lbalt configuration A configuration consists of seven properties, each initialized using an integer value, as follows. Properties powerOn, powerShutdown, powerSleep, and powerStartup represent the power (energy consumption per time unit) of a resource in state On, Suspend, Sleep, and Wakeup, respectively. In the example (see Table 8.2, from top to bottom), property powerOn is 200 watts, powerShutdown 200 watts, powerSleep is 14 Watts, and powerStartup 200 watts. The greater the difference between the powerOn and powerSleep, the more it pays off to shutdown resources. Properties timeStartup and timeShutdown represent the requires time for a resource to either startup

or shutdown. They are both 10 time units in the example. The shorter these times are, the more adequately the load balancer can respond to fluctuations in the incoming load. Finally, property timeOutTime is the time of idling before a resource shuts down to save energy. In the example, it depends on the design dimension *to* in the study (of Section 8.2.2).

Lbalt subprocesses Similar to the process algebra constructs *palt* and *alt*, the *lbalt* contains a number of subprocesses of which exactly one is selected each execution. We define four atomic subprocesses named p_1, p_2, p_3, and p_4 (see Table 8.2) for the load balancer of the example, which map to individual resources next (in Section 8.2.2).

8.2.2 The high-level Resource

The iDSL process (of Table 8.3) consists of four atomic resources, named r_1, r_2, r_3 and r_4. They are the four resources the load balancer will have to choose between. Although not explicitly specified, iDSL is able to derive that these resources are energy-aware resources, viz., the processes of a load balancer are mapped to them (see Section 8.2.3). Note that it is possible to define both energy-aware and regular resources in one iDSL model. In addition to a regular resource, an energy-aware resource consumes energy that can be measured. Also, an energy-aware resource can be in different power states in which its energy consumption and ability to process vary. Section 8.3.2 discusses these power state transitions and the notion of energy consumption in more detail.

Table 8.3: An iDSL resource for a load balancer

```
Section Resource
    ResourceModel r
        decomp { atom r1 rate 1, atom r2 rate 1,
                 atom r3 rate 1, atom r4 rate 1 }
```

8.2.3 The high-level System

As usual, the iDSL system (see Table 8.4) bridges the iDSL process (of Table 8.2) and resource (of Table 8.3) via a mapping. In this mapping, processes p_1, p_2, p_3, and p_4 are assigned to resources r_1, r_2, r_3 and r_4, respectively, all via a FIFO scheduling policy.

Table 8.4: An iDSL system for a load balancer

```
Section System
    Service serv
        Process p
        Resource r
        Mapping assign { (p1, r1) (p2, r2)
                         (p3, r3) (p4, r4) }
                scheduling policy {
                         (r1, FIFO) (r2, FIFO)
                         (r3, FIFO) (r4, FIFO) }
```

8.2.4 The high-level Scenario

The scenario (of Table 8.5) specifies that incoming (service) requests arrive with negative exponentially distributed inter-arrival times with rate 1, at the load balancer. Let predicate $I(t)$ indicate that a request arrived at time t and assume that requests have unique arrival times. Then power set $I : 2^{\mathcal{R}_+}$ indicates, for all moments in time $\mathcal{R}_+$, whether a request arrived at this time or not. For illustration, we show a possible set I, based on random numbers [168]:

$$I = \{0.87,\ 0.91,\ 1.46,\ 2.03,\ 3.54,\ 4.68,\ 5.42,\ 5.52,\ 5.66,\ 7.26,\ 9.61,\ 10.34,\ \cdots\}.$$

We now illustrate the functioning of a load balancer. Assume that incoming requests I arrive at the load balancer, which distributes them over four servers for processing, based on a policy. Let $\mathcal{P}(t) : \mathcal{R}^+ \rightarrow \{S_1, S_2, S_3, S_4\}$ be a load balancer policy that distributes the request that arrived at time t to either server S_1, S_2, S_3 or S_4. Let $S'_m = \{t \in I \mid \mathcal{P}(t) = S_m\}$ be the incoming requests at server m. Hence, $\{S'_1, S'_2, S'_3, S'_4\}$ is a partition of I. For instance, policy $\mathcal{P}$ may decide to distribute the above incoming requests I, as follows.

$$S'_1 = \{0.91,\ 3.54,\ 5.42,\ 10.34,\ \cdots\}, \qquad S'_2 = \{9.61,\ 13.04,\ 13.52,\ \cdots\},$$
$$S'_3 = \{2.03,\ 4.68,\ 5.66,\ 7.26,\ \cdots\}, \qquad S'_4 = \{0.87,\ 1.46,\ 5.52,\ 12.01,\ \cdots\}.$$

To illustrate the large complexity a policy faces, we determine that a policy has to select one out of s^n options after n incoming requests and s servers. For instance, this is already $4^{10} > 1000000$, for only 10 incoming request and 4 servers.

Table 8.5: An iDSL scenario for a load balancer

```
Section Scenario
    Scenario sc
        ServiceRequest serv with distribution exp "1"
```

Table 8.6: An iDSL measure for a load balancer

```
Section Measure
    Measure ServiceResponse times
        using 1 runs of 1500 ServiceRequests
```

8.2.5 The high-level Measure

The iDSL measure (of Table 8.6) specifies that response times of services are retrieved on the basis of one simulation run of 1500 requests. The model checking-based measures of iDSL (see Chapter 4, 5, 6, and 7) are not applied here. Namely, the model is too complex for that because of, among others, the following factors that contribute to the state space: (i) many service requests need to be evaluated to obtain reliable results, due to the exponential distribution for incoming requests; (ii) performance as well as energy information is retrieved; and, (iii) resources have power states in the example, which makes the state space $4^4 = 256$ times as big as the case in which resources all have one state.

8.2.6 The high-level Study

The iDSL study (see Table 8.7) defines the design space, which consists of three design dimensions. These dimensions are used to add variation points to the policy and configuration in the iDSL process. For this purpose, this iDSL process (see Table 8.2) contains three *dspace* constructs:

q the minimum queue size a resource should have, before a new resource is switched on.

nd_res the mechanism used to resolve nondeterminism, either random selection or fixed order of servers selection.

to the time of idling before a resource switches off.

Table 8.7: An iDSL study for a load balancer

```
Section Study
    Scenario sc
        DesignSpace ("q"  {"1" "2" "3" "5" "7"
                           "10" "15" "20" "30"
                           "40" "50" "75" "100" } )
                    ("nd_res" {"random"
                               "(ID/LBNumServers)" } )
                    ("to" {"0" "1" "2" "3" "4" "5"
                           "7.5" "10" "15" "30" } )
```

For instance, design $d = (5, random, 10)$ is the design in which: (i) a new server is turned on when the queue sizes of all currently running servers exceeds 5; (ii) servers automatically shut down after 10 seconds of idling; and (iii) nondeterminism is resolved via a random selection.

8.3 The underlying performance model

We define the semantics of the load balancer (in Section 8.2) via a transformation to an underlying Modest model, which has the following structure (see Figure 8.2). The initial *main* process calls a *main_process*, a *generator*, and four *resource* processes. The *main_process* repeatedly waits for an incoming request from the *generator* followed by a call to the *loadbalancer* process. Finally, a *resource* process can switch back and forth between a *resource* and *resource_off* process, depending on which power state it is in at a given time.

In this section's remainder, we convey the underlying Modest model in more detail, in line with the six sections of iDSL, viz., the Process (Section 8.3.1), Resource (Section 8.3.2), System (Section 8.3.3), Scenario (Section 8.3.4), Measure (Section 8.3.5), and Study (Section 8.3.6).

8.3.1 The underlying Process

In Section 8.2.1, we specified an iDSL process that comprises a load balancer only (see Table 8.2). iDSL transforms it into Modest code (as depicted in Figure 8.2), as follows. Process *main_process* waits for an incoming request from the generator to arrive and then calls process *loadbalancer*. This load balancer

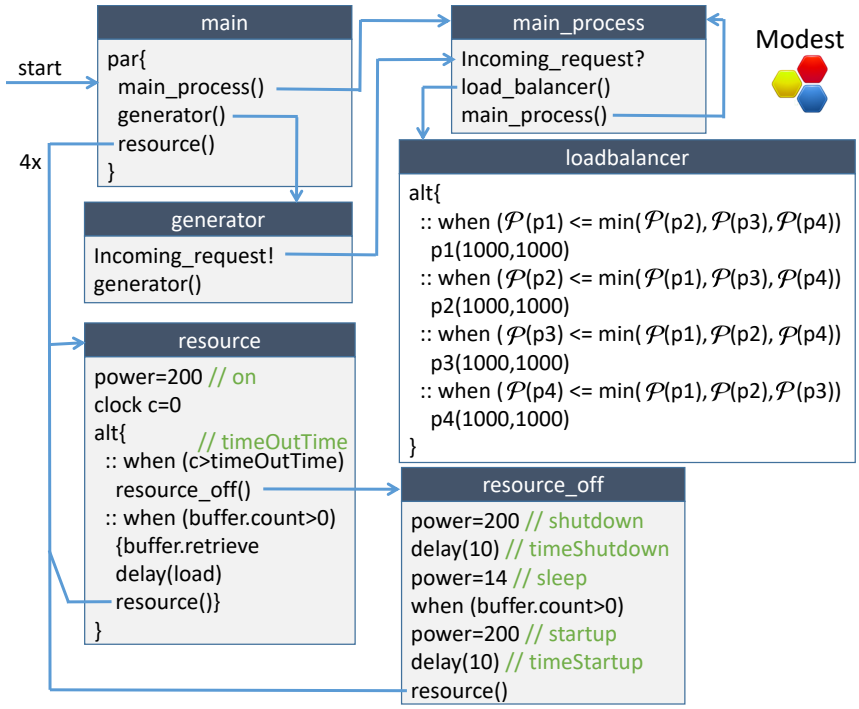


Figure 8.2: The underlying Modest processes, generated for the load balancer

evaluates a given policy $\mathcal{P}$ by separately evaluating this policy for four subprocesses p_1, p_2, p_3, and p_4 and selecting the one that yields the lowest value. These processes are directly mapped to four resources r_1, r_2, r_3, and r_4, respectively.

In the following, we provide the policy grammar to construct valid policies, and the policy semantics to see the meaning of constructed policies by means of a transformation to Modest, respectively.

Policy syntax In Table 8.8, we present the syntax of a policy expression $\mathcal{P}$, which is a subset of the iDSL language. Like many constructs in the iDSL language, a policy expression is recursively defined. This makes it is possible to construct arbitrary complex expressions with only a few constructs, while keeping the transformation to Modest simple by recursively evaluating it. Policy expressions can be something from the categories *state*, *power*, *time* or *math*, but

Table 8.8: The grammar of a policy expression $\mathcal{P}$

$\mathcal{P}$= *state* | *power* | *time* | *math* | *ID* | *numServers* | *queueSize*
state= *stateOn* | *stateSleep* | *stateSuspend* | *stateWakeup*
power= *powerOn* | *powerSleep* | *powerSuspend* | *powerWakeup*
time= *timeWakeup* | *timeSuspend* | *timeOutTime*
math= $\mathcal{P} * \mathcal{P}$ | $\mathcal{P} + \mathcal{P}$ | $\mathcal{P} - \mathcal{P}$ | $\mathcal{P}/\mathcal{P}$ | $\mathcal{P}$ mod $\mathcal{P}$ | *INT* | *random* | *dspace*(*STR*)

Table 8.9: The meaning of the grammar of a policy expression $\mathcal{P}$

State	Indicator functions to see if a server is in a given state or not, e.g., when a server is in state *on*, *stateOn* yields 1 and the others state indicators 0.
Power	The power consumption in a given state (positive integer value).
Time	*timeWakeup* and *timeSuspend* represent the time it takes for the server to go back and forth between states *on* and *sleep*, and *timeOutTime* is the time of inactivity of a server before going to sleep.
Math	Five recursive functions to combine policies via arithmetic operations and create arbitrarily complex policies. Also, it contains a constant integer number, a random number $r \in [0:1]$, and a design dependent number.
ID	A unique number for each server.
numServers	The total number of servers.
queueSize	The instantaneous number of requests in the queue of the server.

can also be an *ID*, *numServers* or *queueSize* construct. In Table 8.9, we informally describe what they mean.

To illustrate how a load balancer policy is parsed, Figure 8.3 provides the concrete syntax (or parse tree) of policy $\mathcal{P}_5$ (cf. Section 8.2.1 and Equation 8.6, the policy of the load balancer used in this chapter. At the top leaf, it shows an arithmetic expression, which uses information of its children for evaluation. Its children are leaves that are evaluated at different stages, viz., the *queueSize* and *stateOn* are dynamically determined every time a request comes in, the dspace q and nd_res (defined in Section 8.2.6) are determined for each design instance, and the *numServers* and 1 are constants that are determined once.

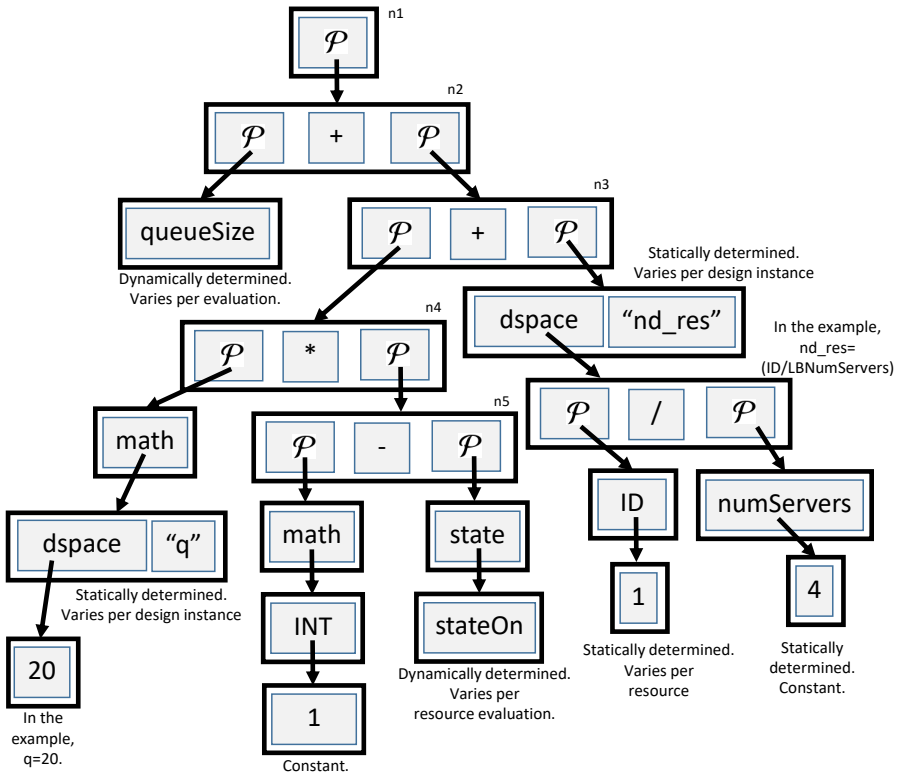


Figure 8.3: The concrete syntax (or parse tree) of the iDSL policy (of Table 8.2), for q=20 and $nd\_res = ID/LBNumServers$.

Policy semantics We illustrate the policy semantics using the evaluation of $\mathcal{P}_5$ as the example. For this purpose, the concrete syntax of Figure 8.3 is transformed into equivalent Modest code for the policy in Table 8.10. This involves the following two steps.

First, a number of assignments are made to evaluate a policy for each of the individual subprocesses, named "evaluate_p1" till "evaluate_p4" in the example. Each assignment involves the evaluation of the concrete syntax of Figure 8.3 for a particular process. This is done recursively for the five mathematical functions that can be used to combine policies, i.e., its both constituents are evaluated first and the result combined using the respective arithmetic operator. Some

Table 8.10: The generated Modest code of the iDSL policy (of Table 8.2), for q=20, and *nd_res*=(ID/LBNumServers)

```
int evaluate_p1=r1_buffer_processid_instance.count+
  (int)20*((int)1-state_r1_on) + 1/4
int evaluate_p2=r2_buffer_processid_instance.count+
  (int)20*((int)1-state_r2_on) + 2/4
int evaluate_p3=r3_buffer_processid_instance.count+
  (int)20*((int)1-state_r3_on) + 3/4
int evaluate_p4=r4_buffer_processid_instance.count+
  (int)20*((int)1-state_r4_on) + 4/4

alt {
:: when ((evaluate_p1 <= evaluate_p2)
  && (evaluate_p1 <= evaluate_p3)
  && (evaluate_p1 <= evaluate_p4)
    p1(1,1)
:: when urgent (...)
     p2(1,1)
:: when urgent (...)
    p3(1,1)
:: when urgent (...)
    p4(1,1)
}
```

functions are dynamically determined and refer to a variable that is declared somewhere in the Modest model, e.g., the indicator function for state *on* for $r1$ is referred to as state_r1_on. Finally, the remaining functions are statically determined and lead to fixed numbers in the Modest code, e.g., a design space value, a constant number, a server ID, and the total number of servers, which are all used in Table 8.10.

Second, the outcomes of the different subprocess evaluations are compared to each other and the subprocess with the lowest evaluation gets executed, cf. the loadbalancer process of Figure 8.2. In Modest, the equation $\mathcal{P}(p_1)$ <= $min(\mathcal{P}(p_2),\mathcal{P}(p_3),\mathcal{P}(p_4))$ is represented as $\mathcal{P}(p_1)$ <= $\mathcal{P}(p_2)$ and $\mathcal{P}(p_1)$ <= $\mathcal{P}(p_3)$ and $\mathcal{P}(p_1)$ <= $\mathcal{P}(p_4)$, as the lower half of Table 8.10 conveys.

Note that either p_1, p_2, p_3, or p_4 always gets executed immediately, because

at least one of the evaluations is the lowest one. Hence, evaluating the policy does not take time in the model, in accordance with assumption $\mathcal{A}10$.

Configuration We define the semantics of the configuration by showing how it is transformed to Modest. In Section 8.2.1, the process of a load balancer (see Table 8.2) is defined that comprises a configuration. This configuration assigns values to seven constants, which are present in the resource and resource_off processes of the transformation to Modest (of Figure 8.2) viz., the resource process contains the powerOn and timeOutTime, and the resource_off process powerShutdown, timeShutdown, powerSleep, powerStartup and timeStartup.

8.3.2 The underlying Resource

In Section 8.2.1, the iDSL resource of a load balancer with four subprocesses (see Table 8.2) is defined. These subprocesses are mapped to four resources in the iDSL system (of Table 8.4). In turn, these resources are used in context of the load balancer. Consequently, iDSL considers the resources to be energy-aware and equipped with power states.

That is, an energy-aware resource is in either power state *on*, suspend (*sd*), sleep (*sl*), or wakeup (*wu*)) at any point in time. Also, the resources exhibit certain behavior as induced by these power states, which we (partially) formally specify next. Figure 8.4 shows the four power states in a diamond shape. It also shows the five possible transitions that can incur between states, as follows.

1. When a server finishes processing its last request and when no new requests arrive in the next TO seconds, the server stays on for TO seconds (Figure 8.4(1a)), suspends for the next 10 seconds ((1b)), and ends in sleep mode ((1c)).
2. When a server is in sleep mode (Figure 8.4(2a)) and a request arrives, it wakes up which takes 10 seconds ((2b)) and then turns on ((2c)).
3. When a server is suspending (Figure 8.4(3a)) and a request arrives, it will finish suspending and then directly wakes up again ((3b)) to turn on ((3c)) eventually.
4. When a server is in sleep mode (Figure 8.4(4)), it remains there as long as no new requests arrive.
5. When a server is turned on (Figure 8.4(5)), it remains turned on when incoming requests arrive frequently enough.

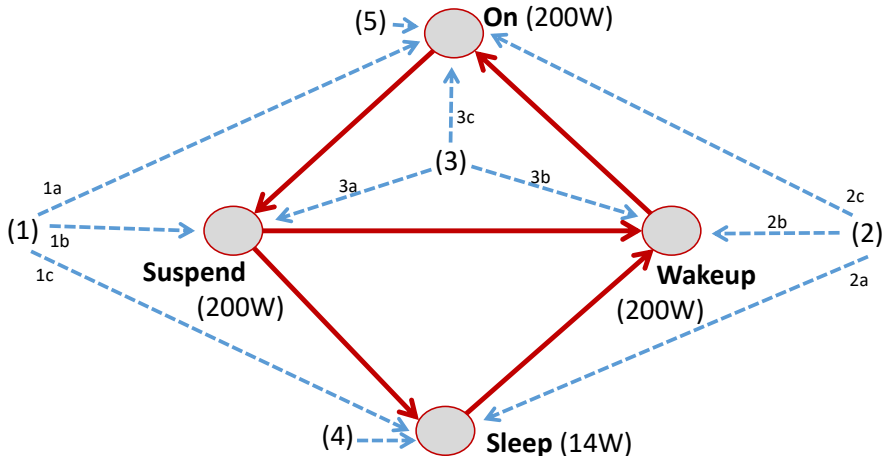


Figure 8.4: The power states of resources (On, Suspend, Wakeup, and Sleep) and the specification elements (1, 2, 3, 4, and 5) applied to them.

In Modest, an energy-aware resource is implemented using two processes (as depicted in Figure 8.2), viz., *resource* and *resource_off*. Resource is the process that is initially executed in state *on* and *power* = 200. Then, either the timeOutTime passes by and the resource gets ready to go to state *sleep* by calling process resource_off, or an incoming job-request is taken from the buffer and processed. In case resource_off is called, the resource consecutively uses 10 seconds to reach state *sleep*, waits for an incoming request, and uses another 10 seconds to go back to state *on* again.

The energy consumption is modeled using the cost/reward construct in Modest and depends on the value of power, viz., for every second that passes the energy consumption is increased by the value of power.

8.3.3 The underlying System

In Section 8.2.3, the iDSL system specifies one service *serv* that maps process p to resource r. In this mapping, atomic processes p_1, p_2, p_3, and p_4, are connected to atomic resource r_1, r_2, r_3, and r_4, all in a FIFO fashion. In Modest, each atomic process calls the respective resource (see Table 8.11).

Table 8.11: The generated Modest code from an iDSL system

```
process p1(real taskload){ r1(taskload) }
process p2(real taskload){ r2(taskload) }
process p3(real taskload){ r3(taskload) }
process p4(real taskload){ r4(taskload) }
```

Table 8.12: The generated Modest code from an iDSL scenario

```
process generator_serv(){
  real expo = Exponential(1);
  clock c; tau {= c=0 =};
  alt{
    :: urgent generator_p!
    :: delay(1 * 0.01)
       tau {= timeouts_p = timeouts_p +1 =}
  };
  when urgent (c >= expo)
    generator_serv()
}
```

8.3.4 The underlying Scenario

In Section 8.2.4, the iDSL scenario specifies an infinite sequence of service requests of service *serv*, which have exponentially distributed inter-arrival times with rate 1.

In Table 8.12, we show the Modest process generator that realizes this behavior (cf. Section 4.3.4 and Table 4.10) using three steps: (i) the Modest library function *Exponential* is used to draw a number according to the exponential distribution; (ii) either the incoming request is forwarded to be processed or a time-out occurs because the system is busy; and, (iii) the delay is performed and the generator starts over again.

We stress that the function *Exponential* can in principle only be used for simulation, which is the only defined performance evaluation technique in the iDSL measure (see Section 8.3.5). However, in case of model checking iDSL is equipped with a discretization step that approximates function *Exponential* to yield a Probabilistic Timed Automata (PTA, [15,129]) model (cf. Section 2.1.5).

8.3.5 The underlying Measure

Section 8.2.5 conveys that measure "ServiceResponse times" is the only performance evaluation technique in the iDSL measure (see Table 8.6). In a simulation run, iDSL does not only retrieve performance information (as in Chapter 4, 5, 6 and 7), but also information about the energy consumption, which is modeled using rewards in Modest (see Section 8.3.2).

We consider the average latency of all requests and average power consumptions to be our prime metrics, as follows. Performance is formally defined as the average latency (AL) of all requests conceivable, as follows.

$$AL = \lim_{n\to\infty} \quad \frac{1}{n}\sum_{i=1}^{n} L_i, \tag{8.7}$$

where L_i is the latency of the i^{th} request. Average power consumed (AP) to keep all four servers running is formally defined, as follows.

$$AP = \lim_{t\to\infty} \sum_{m=1}^{4} \frac{1}{t} \int_0^t (200 \cdot (Pi_m^{on}(s) + Pi_m^{wu}(s) + Pi_m^{sd}(s)) + 14 \cdot Pi_m^{sl}(s))ds, \tag{8.8}$$

where $Pi_m^x(t)$ is the indicator function for resource m being in state x at time t, i.e., $Pi_m^x(t) = 1$ when $P_m^x(t)$, and $Pi_m^x(t) = 0$, otherwise.

8.3.6 The underlying Study

In Section 8.2.5, the iDSL study (see Table 8.6) specifies a 3-dimensional design space with dimensions q, nd_res and to, which consists of $13 \cdot 2 \cdot 10 = 260$ design instances. Consequently, iDSL generates Modest models that vary on design specific aspects and evaluates these individually for each design instance.

8.4 Performance model evaluation

In this section, we briefly illustrate how an iDSL model with a load balancer is evaluated (as depicted in Figure 8.5 and inspired by Figure 5.5). The evaluation comprises the four steps and is similar to the evaluation of Chapter 4, 5, 6 and 7, as follows:

i. Values of design space dimensions q and nd_det are used to compute a policy (as explained in Section 8.3.1), as well as the value of dimension

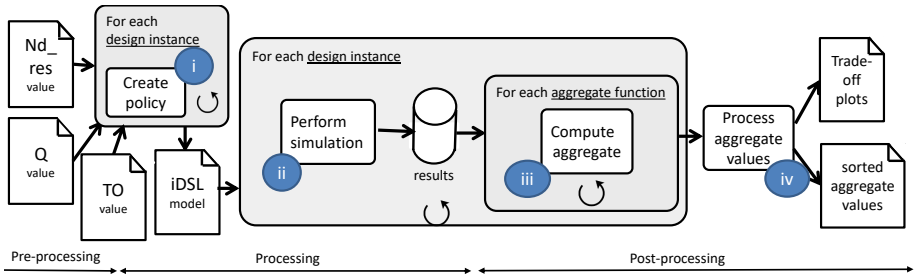


Figure 8.5: The automated iDSL solution chain (see also Figure 5.5) for the load balancer comprising three major steps for each design instance: (i) compute the policy; (ii) perform simulations; and, (iii) computing aggregate functions.

TO to set the resource time-out times. This leads to an iDSL model that contains the policy and time-out times for resources.

ii. the actual evaluation takes place via a MODES simulation run.

iii. the results are post-processed to extract performance and energy consumption numbers from them, i.e., the average latency (of Equation 8.7) and the average power consumption (of Equation 8.8).

iv. these post-processed results are used to generate trade-off plots. These plots may enhance decision making by conveying the price of gaining performance at the cost of energy consumption, and vice versa.

8.5 Performance results

In the following, we show what lessons the results of the evaluated designs teach us about policies (in Section 8.5.1), followed by an assessment of their validity (in Section 8.5.2).

8.5.1 Lessons learned

In Figure 8.6 and 8.7, we show the total power consumption (on the y-axis) versus the mean response time (x-axis), for many designs. The designs have been evaluated using both the iDSL and Anylogic implementation (as introduced in Section 8.1, Step 5). However, for practical reasons, we have only

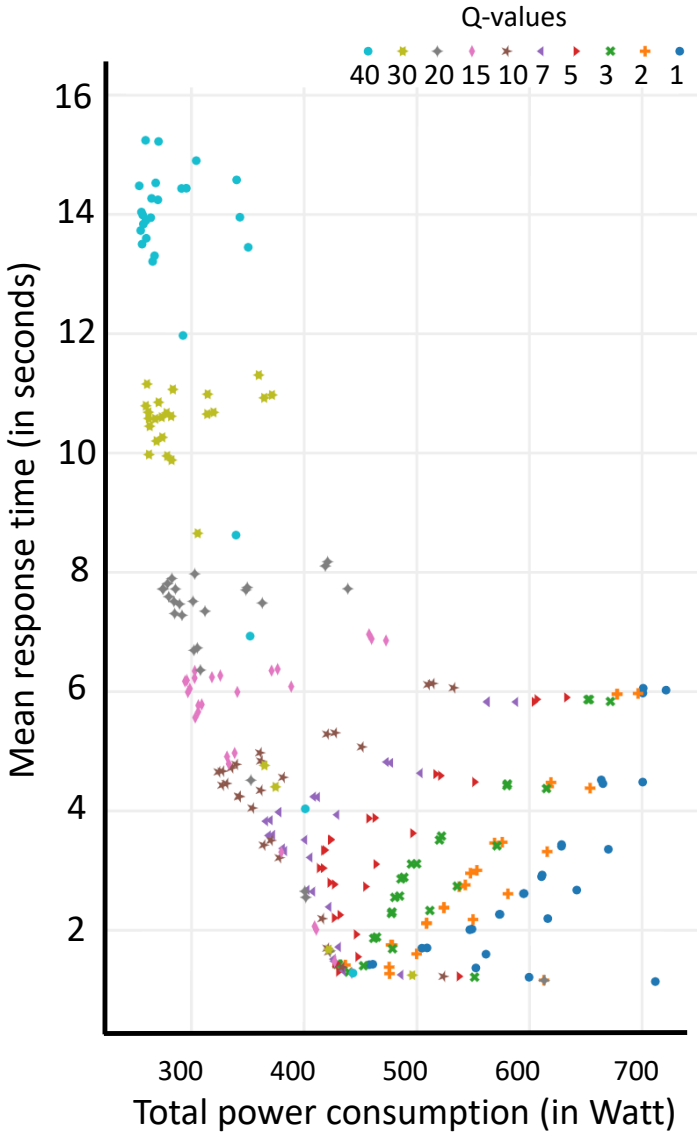


Figure 8.6: Average latency and power consumption outcomes for many designs, which are classified by their value for dimension queue (Q) by plotting them the same color, as follows: designs with q=1 are Dark-blue, q=2 Orange, q=3 Green, q=5 Orange-red, q=7 Orchid, q=10 Copper, q=15 Magenta rose, q=20 Silver, q=30 Pear, and q=40 Light-blue.

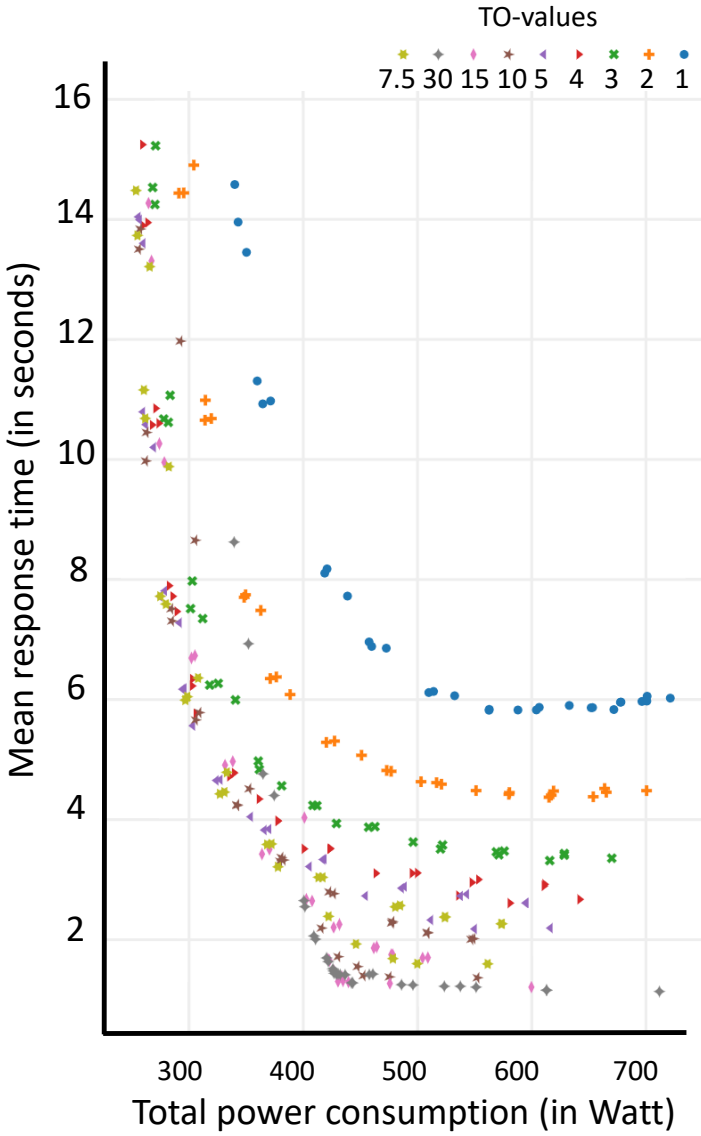


Figure 8.7: Average latency and power consumption outcomes for many designs, which are classified by their value for dimension time-out (TO) by plotting them the same color, as follows: designs with TO=1 are Dark-blue, TO=2 Orange, TO=3 Green, TO=4 Orange-red, TO=5 Orchid, TO=10 Copper, TO=15 Magenta rose, TO=30 Silver, and TO=7.5 Pear.

plotted the results of the Anylogic implementation using the online graphing utility Plotly [161]. In the end, this does not make a difference as the iDSL and Anylogic results are similar, which we will demonstrate in Section 8.5.2.

Figure 8.6 and 8.7 illustrate the effect of adjusting the parameters in the policy for various values of queue threshold $q \in \{1, 2, 3, 5, 7, 10, 15, 20, 30, 40\}$ and the time-out $TO \in \{1, 2, 3, 4, 5, 7.5, 10, 15, 30\}$. Normally, a system with a low power consumption and a low latency is preferred, which makes the designs located on the bottom-left of the figures the “better" ones. In both figures, the same points are plotted, but the basis on which they are colored is different, as follows.

The effect of q. In Figure 8.6, designs with equal values for q are marked with the same color, as indicated in the caption of the figure. As anticipated, it shows that for high values of q, e.g., $q = 40$ (light-blue), with all average power consumptions below 400 watts, lead to low energy consumption but also poor performance. In contrast, low values of q, e.g., $q = 1$ (dark-blue), with all mean response times below 7 seconds, lead to high performance but at the price of an increased energy consumption. Hence, the selection of a value for q, the minimum queue size of the currently running resources before a new resource is switched on, can also be seen as a preference for either performance (i.e., a low q-value) or energy consumption (i.e., a high q-value).

The effect of TO. In Figure 8.7, designs that share the same TO value are colored the same, again indicated in the caption of the figure. It can be seen that the position of the efficiency frontier depends on the time-out value TO. Concretely, the higher the selected value of TO, the better the combined performance and energy consumption turns out. For instance, for $TO = 30$ (silver) many designs convey low response times and a low energy consumption. In contrast, for $TO = 1$ (dark-blue) both the response times and energy consumption are significantly higher. Hence, the system designer has to select a high value for TO to ensure Pareto optimality. Based on these observations, we reason that it is costly to switch resources off in the example of this chapter. This can be caused by a combination of the following factors:

i. High state transition times *TimeStartUp* and *TimeShutDown*: makes it time consuming to turn resources off and on,

ii. A high power in states *Suspend* and *Wakeup*: a resource cannot process in these states but yet uses the same power as in state *On*,

iii. A high incoming request rate: it does not pay of to shut resources down when sufficient requests keep arriving.

To conclude, we have gained two insights from the results (of Figure 8.6 and 8.7). Parameter q of the design space can be used to resolve the trade-off between performance (a low q-value) and energy consumption (a high q-value). Also, we found that the system designer has to select a high value for parameter TO to ensure Pareto optimality in the example of this chapter.

8.5.2 Validity of the results

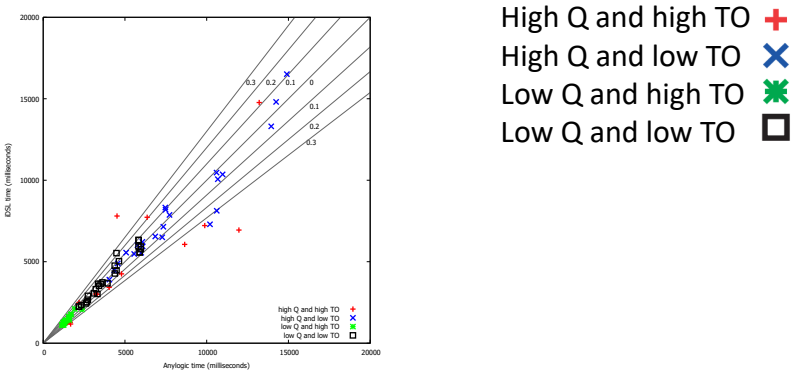
To validate the iDSL implementation of a load balancer as presented in this chapter, we have simultaneously implemented the load balancer [204] in Anylogic [4, 25]. We have compared the outcomes, i.e., the performance and energy consumption outcomes, for many different designs. The following distance measure, which returns the ratio differences, has been used to compare outcomes:

$$\delta(v_1, v_2) = \max\left(\frac{v_1}{v_2}, \frac{v_2}{v_1}\right) - 1, \tag{8.9}$$

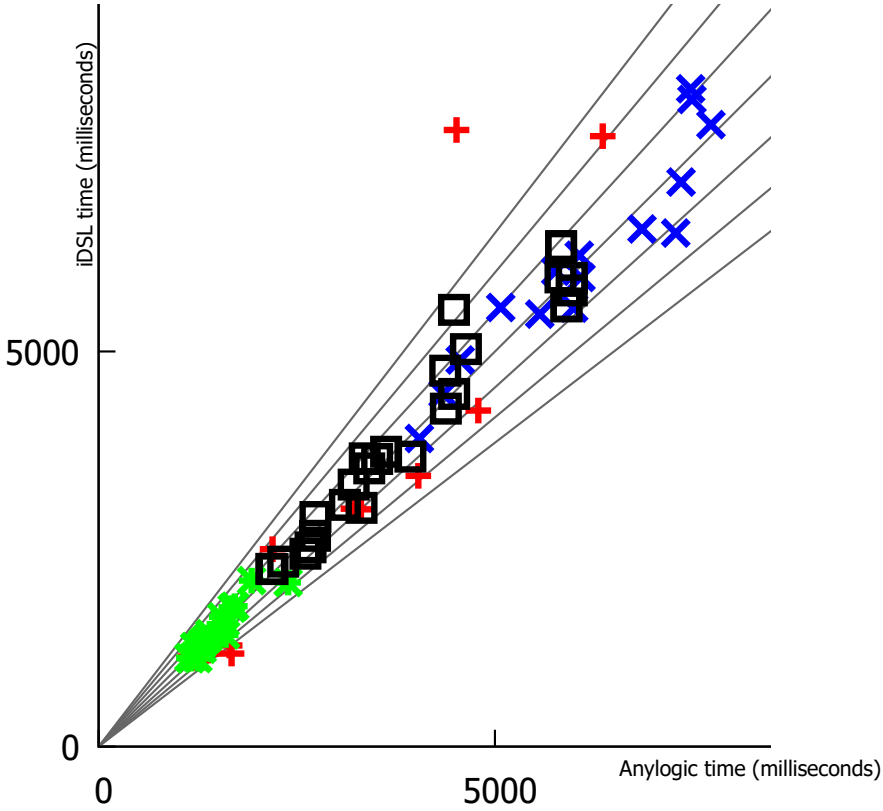
where v_1 is the outcome of implementation 1, and v_2 of implementation 2.

The measure is designed to behave like a metric, viz., $\delta(v, v) = 0$, $\delta(v_1, v_2) = \delta(v_2, v_1)$ (symmetry), and $\delta(av_1, av_2) = \delta(v_1, v_2)$ (scalability). However, the triangular property $\delta(v_1, v_2) + \delta(v_2, v_3) \geq \delta(v_1, v_3)$ does not hold. In the following, we compare the outcomes for designs in which nondeterminism is solved by randomness and fixed order, respectively, after which we draw some general conclusions.

Resolving nondeterminism with randomness Figure 8.8 shows the experimental latency outcomes of iDSL (on the y-axis) and AnyLogic (on the x-axis) using randomness to resolve nondeterminism. Figure 8.8(a) provides a full, albeit small overview, whereas Figure 8.8(b) focuses on the smaller latencies. The designs are categorized in four categories: designs with $q > 15$ and $TO > 4$ (in red), designs with $q > 15$ and $TO \leq 4$ (in blue), designs with $q \leq 15$ and $TO > 4$ (in green), and designs with $q \leq 15$ and $TO \leq 4$ (in black). Note that the distance δ is visualized around the diagonal for values 0, 0.1, 0.2 and 0.3. The

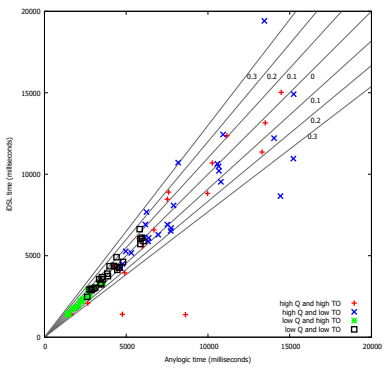


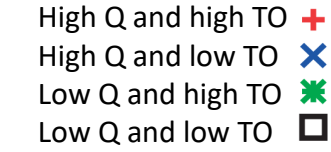
(a) Full overview



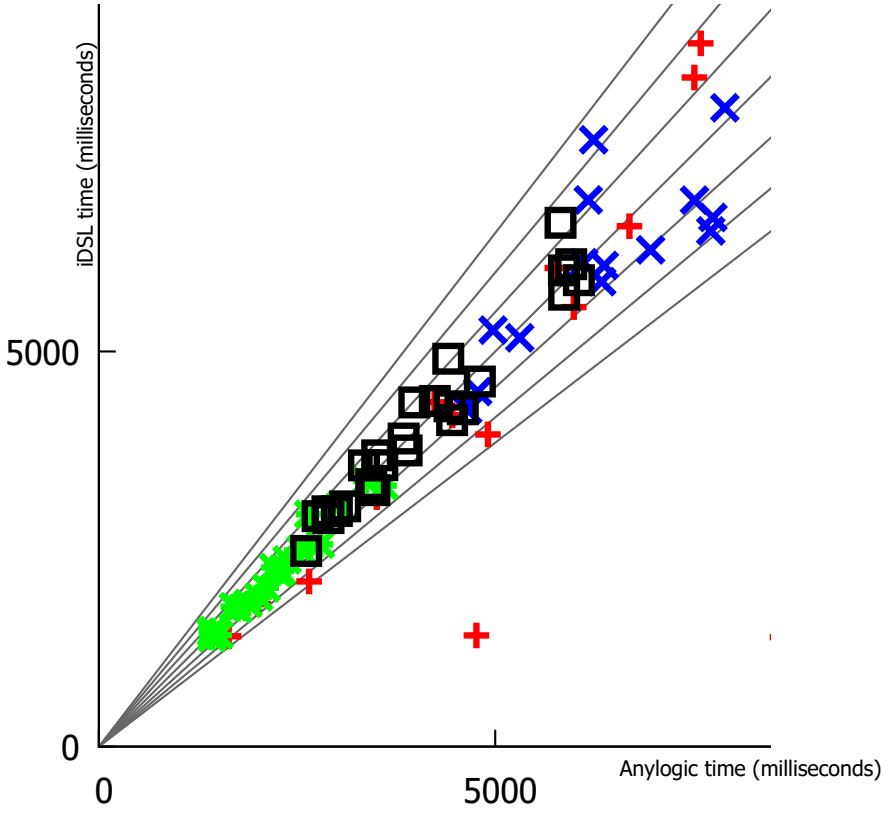
(b) Zoomed in graph

Figure 8.8: Comparison of the iDSL (on the y-axis) on AnyLogic (on the x-axis) results for many designs using randomness to resolve nondeterminism.





(a) Full overview



(b) Zoomed in graph

Figure 8.9: Comparison of the iDSL (on the y-axis) on AnyLogic (on the x-axis) results for many designs using a fixed order to resolve nondeterminism.

closer designs are to the diagonal, the better the results of both implementations match.

We observe that designs with a low Q-value (in green and black) as well as a low TO-value (in blue) are positioned close to the diagonal, i.e., their values for δ are smaller than 0.3. Contrarily, designs with a high Q-value and high TO-value (in red) display some outliers, especially in Figure 8.8(a), which we will discuss later.

Resolving nondeterminism with fixed order Figure 8.9 also shows the experimental latency outcomes of iDSL and uses fixed order to resolve nondeterminism. As in Figure 8.8, Figure 8.9(a) and 8.9(b) provide a full and zoomed-in overview, respectively. Also, the same four categories of designs are the same..

We observe that a few designs with a high Q-value and high-T value (in red) are far from the diagonal, but in addition to Figure 8.8(a) some designs with $q > 15$ and $TO \leq 4$ (in blue) score greater than 0.3 on δ metric.

In short, the results of the iDSL and Anylogic implementation match for a vast number of designs. However, especially for designs with a high Q-value, TO-value, and/or randomness to resolve nondeterminism, we observe different outcomes. We illustrate how each ones of them increases the state space, as follows:

(i) Q-value: the effect of a high Q-value is can be seen best when there are many requests in the system;

(ii) TO-value: the effect of a high TO-value can only be seen when the system is mostly empty; and,

(iii) Resolving nondeterminism randomly: causes unpredictive and thus more complex behavior than resolving nondeterminism in a fixed order way.

Note that a large state space can make both implementations easily differ from each other, especially when discrete-event simulation is used as the evaluation technique.

We propose the following steps to for further investigation that address the increased state space: (i) apply more and longer simulations runs per design; (ii) inspect the outliers individually; and, (iii) use different values for the rate parameter. This further investigation is beyond the scope of this thesis.

8.6 Conclusion

In Chapter 4, 5, 6 and 7, we applied iDSL to the domain of iXR systems. Since we would like to know whether iDSL can be used well in other domains, we applied iDSL to a different case study on load balancers in this chapter. In this case study, iDSL is not only used to evaluate the performance of load balancer, but also to gain insight in energy consumption, so that an optimal trade-off between performance and energy can be obtained.

The implementation of the load balancer takes three steps, as follows. First, we have constructed a model to evaluate the performance and energy consumption of load balancers. This model defines a powerful policy language that decides to which server jobs are assigned, achieved by observing the system variables that can be relevant to the load balancer, e.g., queue sizes of servers.

Second, to evaluate the performance and energy trade-off of many different policies, we have implemented two load balancers based on exactly the same specification, in iDSL [202–206] and in AnyLogic [4].

Third, to evaluate these many policies and specify how incoming requests are distributed over servers, a 3D design space has been defined with the following parameters: (i) q is the queue threshold before new servers are switched on; (ii) nd_res: the way nondeterminism is resolved in the policy; and, (iii) TO: the time of idling before a server is shut down. A policy design is a combination of valuations for these parameters and is used to evaluate one policy.

The iDSL implementation evaluates many policies in a useful way, as follows. Parameter q, the queue threshold for switching servers on, can be used to put emphasis on either performance or power consumption, viz., low q values leads to good performance, while higher q values reduce energy consumption. Parameter TO, the idling time of servers before putting them to sleep, determines the position of the so-called Pareto optimal frontier. In our specific case, a higher TO value improves both performance and power consumption. Hence, switching a server off is generally not a good idea in this context.

For validation, we have implemented a load balancer in iDSL and Anylogic, and evaluating designs using it has led to useful performance and energy consumption outcomes. For validation, we have compared the iDSL implementation with an Anylogic implementation, yielding similar results for the majority but not all designs. To have even more trust in the validity of the iDSL model, it is therefore recommended to perform extra measures, such as longer and more simulation runs as well as investigating the outliers.

We conclude from this that iDSL can be used in a different domain, but that validating the implementation requires caution.

CHAPTER 9

Conclusions

In this thesis, we have presented a method for performance evaluation of service-systems. This conclusion provides a brief overview of the performance evaluation process and its overall goal, a number of experiments to reach this goal, an evaluation with respect to reaching this goal, the conclusion of the thesis as a whole, and new ideas for future work.

The performance evaluation process

To gain insight in the performance of embedded systems, we have proposed a framework for performance evaluation. It consists of four components that are related to performance. A *high-level performance model* (cf. Indicator $\mathcal{I}1$ of Section 1.3) is obtained by modeling the performance of a system using a high-level language. This model is transformed into a *underlying performance model* (cf. Indicator $\mathcal{I}2$), which adheres to a widespread formalism, e.g., Stochastic Timed Automata (STA, [32,82]). This formalism enables *performance evaluation* (cf. Indicator $\mathcal{I}3$) using a variety of techniques, and yields *performance results* (cf. Indicator $\mathcal{I}4$).

In the performance evaluation process, an underlying formalism and evaluation technique are selected. Given a formalism, only certain evaluation techniques are applicable, e.g., discrete-event simulation can be applied to an STA model, but model checking cannot. The right underlying formalism and evaluation technique are key to evaluating the performance, because they affect three essential criteria, viz., the variety of metrics that are supported, the wall-clock time and computer memory needed, and the quality (i.e., the accuracy, precision and information) of the results.

State-of-the-art

In the literature, many toolsets have been proposed and created to facilitate the performance evaluation process. We have evaluated fifteen toolsets (cf. Appendix B.1 till B.15) using the four indicators above, leading to an indicator-wise overview of the state-of-art, as follows:

- $\mathcal{S}1$ The high-level performance models of toolsets tend to be generic and not tailored to a specific domain, which makes them easy to transform into underlying models. However, this comes at the price of an increased modeling effort and also makes it harder to communicate and understand a model. Many toolsets do not support the Y-chart philosophy and Design Space Exploration (DSE) explicitly.
- $\mathcal{S}2$ Their underlying performance models tend to contain a great level of detail, which makes them widely applicable.
- $\mathcal{S}3$ The toolsets generally use discrete-event simulation to evaluate the performance model, but generally lack model checking support for exact results. Also, performance evaluation is often not fully automated, forcing the system designer to perform labor-intensive steps, e.g., model calibration and evaluating many designs.
- $\mathcal{S}4$ The results are often not visualized automatically.

Goal

In this thesis, we have built on the literature by extending the four state-of-the-art aspects above. For this purpose, we first constructed an indicator-wise overview of goals, as follows:

- $\mathcal{G}1$ An expressive, yet concise high-level modeling language tailored to service systems should be constructed, equipped with an integrated Development Environment (IDE). The language should be transformable into multiple formalisms, and support the Y-chart philosophy and model calibration.
- $\mathcal{G}2$ The high-level performance model should be relatively abstract. Therefore, only a small subset of the underlying language can be generated. The transformation should be customizable to enable or disable nondeterministic and probabilistic choices.

$\mathcal{G}3$ The performance evaluation should use multiple evaluation techniques on different formalisms, evaluate multiple designs, and be fully automated. That is, a system designer creates a model and, after a limited amount of time, performance results are automatically returned.

$\mathcal{G}4$ These results can be numerical, e.g., latency values and utilizations, which moreover should be visualized, e.g, as with latency bar charts and latency breakdown charts.

Experiments

In this thesis, we have performed five experiments to bridge the gap between the state-of-the-art and the goal of this thesis. Each experiment has been described in one chapter of this thesis, viz., Chapter 4, 5, 6, 7 and 8, respectively. In the following, we describe what each experiment has contributed regarding the goal of the thesis.

In Chapter 4, iDSL was constructed. The iDSL language is an expressive, yet concise high-level language tailored to service systems. Language instances are transformed into Modest models under the hood. Modest models are evaluated using simulations, e.g., yielding latencies and utilizations, and model checking to obtain absolute bounds. iDSL turns the evaluation results into visualizations, such as latency bar charts and latency breakdown charts. Hence, iDSL fully automates the trajectory from iDSL models to all resulting performance numbers and visualizations.

In Chapter 5, iDSL was extended by conducting a case study on biplane interventional X-ray (iXR, [158,159]) systems. We have added automated support for model calibration using measurements to make the models more realistic. Also, performance results can automatically be combined using so-called aggregate functions. Finally, iDSL can automatically generate trade-off graphs to compare designs.

In Chapter 6, iDSL was extended with a new evaluation technique based on iterative probabilistic model checking. This technique allows for exact latency distributions to be obtained. However, it faces the state space explosion problem, which was dealt with using manual model simplifications.

In Chapter 7, the evaluation technique of Chapter 6 was extended by adding automated model simplification techniques to it. Additionally, the technique has been made more efficient by combining it with light-weight evaluation techniques, such as basic estimates and discrete-event simulation.

In Chapter 8, iDSL was used to model and evaluate a load balancer in a

case study. For this purpose, iDSL has been extended with the notion of energy-efficient resources, measurable energy consumption, and a load balancing construct that comes with a load balancing policy.

Evaluation

In this section, we evaluate to what extent the experiments (of Chapter 4, 5, 6, 7 and 8) contributed to the goal of this thesis (cf. Section 1.4), as follows.

- $\mathcal{G}1$ The *high-level performance model* that has been constructed (in Chapter4) and extended (in Chapter 5, 6, 7 and 8) provides an expressive, concise model of service-oriented systems, which explicitly supports the Y-chart philosophy. Additionally, model calibration has been added (in Chapter 5), been taken advantage of and validated (in Chapter 5, 6 and 7).
- $\mathcal{G}2$ The *underlying performance model* is automatically obtained via a transformation from the high-level model into a Modest model. This allows the enabling and disabling of properties like nondeterministic and probabilistic choices.
- $\mathcal{G}3$ The *performance model evaluation* comprises a fully automated approach, which includes the evaluation of multiple designs and generation of visualizations, from a high-level model to final results. Besides support for simulation we have added exhaustive analysis using both Model Checking Timed Automata (MCTAU, [24, 83]), UPPAAL (in Chapter 4), and MCSTA (in Chapter 6 and 7). In two experiments (cf. Chapter 5 and 8), however, we have resorted to simulations only, due to the high-complexity of the models. Despite the use of model simplification techniques that reduce the quality of the results, performance evaluation took in the order of minutes for simulation (e.g., in Chapter 5) and many hours for model checking, such as in Chapter 6 and 7.
- $\mathcal{G}4$ Initially, the *performance results* comprised many measures of interests (cf. Chapter 4), but later only latency distributions were supported (cf. Chapter5, 6, 7 and 8). Hence, only latency distributions, which are the key for the use case and service-oriented systems, can be obtained via evaluation techniques other than simulation.

In short, an automated solution chain from a high-level model to comprehensive performance results has been established. The high-level model facilitates the modelling of complex systems. Scalability remains to be an issue, even though

we provide a high-level language that encourages creating simpler models and model simplification techniques. This scalability problem originates from a complex model, the desired quality of the results, and computational resources and wall-clock time at hand.

Future work

In future work, transforming an existing language for performance evaluation into the iDSL language might be valuable. It will put the expressibility of iDSL to the proof and enables evaluation techniques of iDSL to become applicable to the instances of the existing language. We particularly find the business process domain interesting, e.g., business process modeling notation (BPMN, [148, 173]), because business processes are similar to services and widespread in literature.

When used commonly, the iDSL language might require some extensions of which we mention a few. For processes, we foresee split, merge and repeat operators to be useful. Also, atomic tasks that require multiple simultaneous resources seem an helpful addition to the language. For resources, we consider adding non-renewable resources that can only be used once, as in production systems with goods. For the system, a variety of scheduling policies and dynamic mappings can be implemented. Although some of the above might introduce deadlocks as a side effect, finding ways to resolve or avoid these deadlocks is an interesting challenge.

During the development of iDSL, we primarily focused on returning latency distributions, which was driven by the current literature as well as the conducted use cases on Medical Imaging Systems. On top of this, it might be interesting to investigate whether additional measures of interests can be obtained via evaluation techniques other than simulation.

Regarding Design Space Exploration, iDSL has been equipped with an exhaustive method the evaluates each individual design of the design space. For optimality, it can be interesting to add more advanced way to explore the design space to iDSL, such as hill climbing-based methods [98, 216]. For this purpose, the performance queries of iDSL (see Section 6.2.7 and 6.2.8) are a useful mechanism to compare designs on one or more aspects.

Finally, recent work on the Modest toolset [84] in which the overlap of the models for different iterations of time is exploited, appears to be promising in dramatically reducing the evaluation time.

Appendices

APPENDIX A

Performance evaluation process assessment

In Section 1.3, four indicators of the performance evaluation process were presented, which directly correspond to the prediction-specific components of the performance evaluation process (see Figure 1.1(b)).

In this appendix, we differentiate these indicators, which leads to more specific performance indicators. We have included a variety of specific indicators to get insight in what the main four indicators are about, but will only address a subset of all these indicators in this thesis.

For illustration, Figure A.1 gives a visual overview of the indicators of concern and how they depend on each other (as depicted by arrows). Indicators without dependences contribute indirectly to the performance evaluation process, such as Tooling (cf. Indicator $\mathcal{I}3.3$)

A.1 The high-level performance model

The indicators for the high-level performance model address, among others, the ease of modeling, supported features, and familiarity of the language, as follows.

$\mathcal{I}1.1$. Ease of modeling: the expressibility of the modeling language, the overall size of the generated models, and the amount of human effort needed to create a high-level performance model, e.g., the presence of an integrated Development Environment (IDE) that makes suggestions and validates created models.

$\mathcal{I}1.2$. Y-chart philosophy support: the degree in which the performance modelling language follows the Y-chart philosophy [13,14] by supporting, and

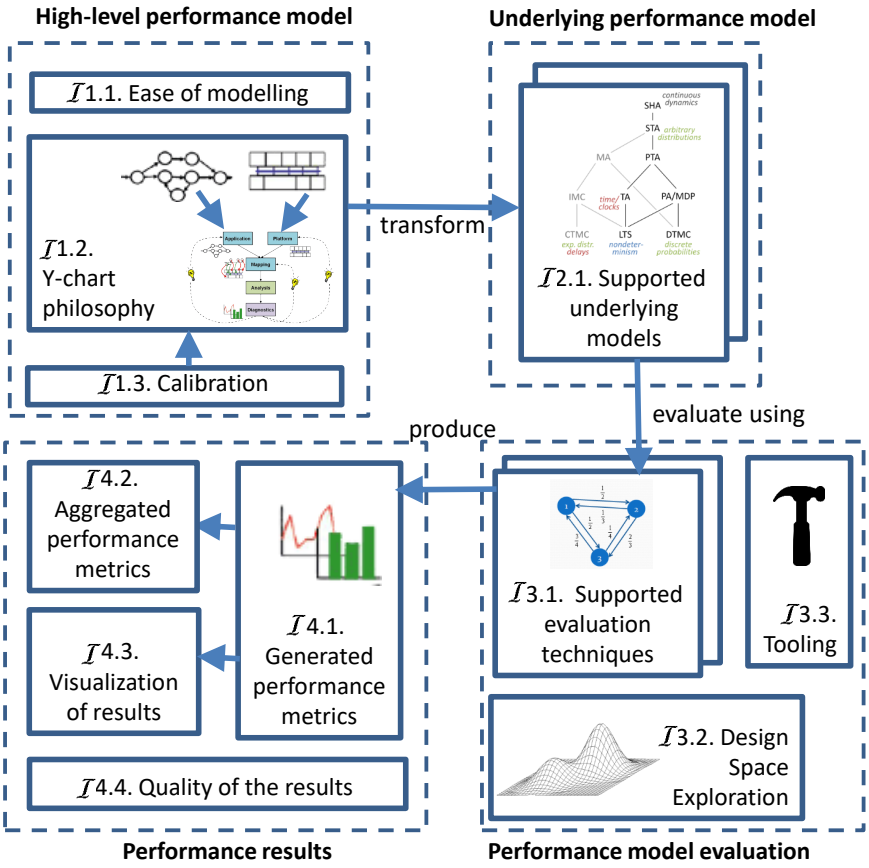


Figure A.1: The indicators of concern and their dependences

separating, the specification of applications, platforms, and mappings.

$\mathcal{I}$1.3. Calibration: the available techniques to calibrate the high-level performance model, based (on as few as possible) real measurements that have been performed on an existing system.

$\mathcal{I}$1.4. Communicability: the extend in which the high-level performance modelling language enhances communication between stakeholders.

$\mathcal{I}$1.5. Modeling freedom: the assumptions and restrictions of the high-level performance modelling language that affect which range of underlying models can be created from it.

$\mathcal{I}$1.6. Familiarity of the performance modelling language: supporting a language that is known by a large group of users, e.g., C++ or Java programmers.

Indicators $\mathcal{I}$1.1- $\mathcal{I}$1.3 are addressed in this thesis. Contrarily, Indicator $\mathcal{I}$1.4 is of minor concern, because the system designer is not going to directly use the language yet. Indicator $\mathcal{I}$1.5 and $\mathcal{I}$1.6 are also not addressed, viz., they inherently lead to a generic performance model that normally is harder to analyze.

A.2 The underlying performance model

The indicators for the underlying performance model are mainly concerned with which underlying models can be generated from the high-level performance model and their properties. Therefore, Indicator $\mathcal{I}$2.2 till $\mathcal{I}$2.5 can be individually applied to all underlying models that can be generated (of Indicator $\mathcal{I}$2.1).

$\mathcal{I}$2.1. Underlying performance models supported: the underlying performance models the high-level performance model can be transformed to. In Section 2.1, an extensive overview of formalisms is provided.

$\mathcal{I}$2.2. Ease of transforming the high-level performance language to the underlying performance language: the ease with which the high-level performance language can be transformed into the underlying performance language, e.g., the size and complexity of the transformation in programming code, as well as the effort and time to perform such a transformation.

$\mathcal{I}$2.3. Separation of concerns between the high-level performance modelling language and underlying performance modelling language: the ease with which either one can be replaced without affecting the other.

$\mathcal{I}$2.4. Easy of modelling, communicability, and modelling freedom of the underlying performance model: as with Indicator $\mathcal{I}$1.1 till $\mathcal{I}$1.6 (recursion).

$\mathcal{I}$2.5. Understandability: the degree in which the generated underlying model can be understood without knowing the high-level-performance model, e.g., by using comments, hierarchical functions and libraries.

Indicator $\mathcal{I}$2.1 is addressed in this thesis. Namely, the more underlying models are supported, the more performance evaluation techniques can be used. This leads, in turn, to a larger variety of results. In contrast, Indicator $\mathcal{I}$2.2 till $\mathcal{I}$2.5 are not addressed, because they are about the qualities of the underlying language, which are only relevant while creating the transformations initially.

A.3 High-level performance model evaluation

The following indicators of the high-level performance model evaluation are concerned with generating results based on an underlying model.

$\mathcal{I}$3.1. Supported performance evaluation techniques: the range of methods used to evaluate the performance, such as simulation and model checking. In Section 2.2, an in-depth overview of these techniques is provided.

$\mathcal{I}$3.2. Design Space Exploration (DSE. [13, 14, 104]): the possibility of conveniently evaluating many different designs. We make a distinction between brute force methods that evaluate all designs and heuristics-based methods that systematically evaluate a subset of all designs, such as hill climbing [174].

$\mathcal{I}$3.3. Tooling: the amount of human effort needed to evaluate a high-level performance model.

$\mathcal{I}$3.4. Interactivity: the ability of the tool to interact with the user while the tool is performing the performance evaluation process, e.g., the user might want to adjust parameters after seeing intermediate results.

$\mathcal{I}$3.5. Execution time: the amount of wall-clock time needed to evaluate the performance model.

$\mathcal{I}$3.6. Scalability: the degree to which the performance evaluation time increases in terms of an increase in size or complexity of the performance model.

$\mathcal{I}$3.7. Generation of executable code: The ease with which executable software to be run on the real system can be derived from the performance model.

Indicators $\mathcal{I}$3.1 till $\mathcal{I}$3.6 are addressed in this thesis, opposed to the generation of executable code (Indicator $\mathcal{I}$3.7), which we found to be too implementation dependent in prior work [207] (cf. Appendix C).

A.4 Performance results

The following indicators are concerned with what performance results can be generated, how these results are delivered, and how much these results reflect reality.

$\mathcal{I}$4.1. Generated performance metrics: the generated measures during performance model evaluation, e.g., service latency distributions, resource utilizations, average queue sizes and number of time-outs.

$\mathcal{I}$4.2. Aggregated performance metrics: the provided mechanisms to combine generated measures (of Indicator $\mathcal{I}$4.1) into newly aggregated measures, e.g., the latency per pixel combines the image latency and number of pixels per image.

$\mathcal{I}$4.3. Visualization of results: means to present performance results (of Indicator $\mathcal{I}$4.1) graphically and/or in diagrams in a graphical, understandable way to be interpreted by humans, such as a graph containing multiple latency cumulative distribution functions.

$\mathcal{I}$4.4. Quality of the performance results: a combination of (i) accuracy, the difference between the generated results and the true results; (ii) precision, the degree to which the generated results are consistent when performance model evaluation is repeated; and (iii) information, the amount of information the results provide. In Section 2.3.2), we provide an extensive explanation of these three aspects.

$\mathcal{I}$4.5. Communicability: the extend to which results enable communication between stakeholders, e.g., the system designer explaining what the performance results mean in reality to future system users.

The actual results artifacts generated (of Indicators $\mathcal{I}4.1$ till $\mathcal{I}4.4$) are addressed in this thesis. Contrarily, Indicator $\mathcal{I}4.5$ is of less concern, because the generated results are normally not directly shown to the system user (yet).

APPENDIX B

Performance evaluation toolsets

In this appendix, we compare fifteen performance evaluation toolsets (in Appendix B.1 till B.15), ordered alphabetically by name, that are well-known in the literature to determine the state-of-the-art regarding the performance evaluation process (as depicted in Figure 1.1(b)). The outcomes of this comparison are summarized in Section 1.3, and subsequently used to assess which extensions on what aspects are desirable in Section 1.4.

The comparison of the fifteen toolsets takes place on the basis of Indicators $\mathcal{I}1$ till $\mathcal{I}4$ as introduced in Section 1.3 and elaborately explained in Appendix A They correspond to the high-level performance model, underlying performance model, performance evaluation, and performance results, respectively.

B.1 Coloured Petri-Net tools

Coloured Petri-net tools (CPN tools, [42, 108, 169]) is a toolset for editing, simulating and analysing Coloured Petri-Nets, which are a backward compatible extension of the concept of Petri nets. CPN preserve useful properties of Petri nets and at the same time extend initial formalisms to allow the distinction between tokens [108].

$\mathcal{I}1$ High-level performance model: CPN tools comes with a Graphical User Interface (GUI), which incorporates the most recent human-computer interaction concepts. Using this GUI, colored Petri-Nets can be designed graphically. The Y-chart philosophy [13, 14] is not explicitly supported, but can be implemented fairly easy by creating distinct Petri-Net models for the process and resource, and connecting them to represent a mapping. Model Calibration can be accomplished by representing measurements as discrete uniform distributions.

$\mathcal{I}2$ Underlying performance model: CPN tools uses colored Petri-Nets (cf. Section 2.1.9) under the hood.

$\mathcal{I}3$ Performance evaluation: CPN tools supports automated simulations to browse through an execution trace, as well as state space analysis to quickly generate (part of) the state space. The latter leads to boundedness properties and liveness properties. Design exploration is implicitly supported by using variables. CPN provides a fully automated approach.

$\mathcal{I}4$ Performance results: Performance evaluation yields end-to-end delays, and average queue lengths, e.g., represented by the individual tokens on a place, and resource utilizations CPN tools outputs the results conveniently albeit without visualizations. The results are of limited accuracy and precision as they result from simulations.

In short, CPN tools comes with a **GUI** to create colored Petri-nets, which can be seen as a low-level formalism (e.g., due to their elegantly simple syntax and semantics, as well as their general applicability). **Simulation** is the prime method of analysis and results into **end-to-end delays** and average queue lengths. These results are not presented visually.

B.2 Differential Equation Analysis of Layered Queuing Networks

Differential Equation Analysis of Layered Queuing Networks (DiffLQN, [47, 210]) is a tool for the analysis of networks using Ordinary Differential Equations (ODE).

$\mathcal{I}1$ High-level performance model: DiffLQN represents its high-level model as a Layered Queueing Network (LQN, cf. Section 2.1.9), entered in text format in an IDE with input validation. “layers”. Hence, the language explicitly supports the Y-chart philosophy.

$\mathcal{I}2$ Underlying performance model: A LQN gets transformed into a PEPA process-algebra model (as presented in Appendix B.11) or a Matlab compatible form [141], which can be solved by a Matlab ODE solver.

$\mathcal{I}3$ Performance evaluation: The underlying PEPA model is analyzed using the ODE setting in the PEPA tools. Design Space Exploration (DSE) is future

work. Namely, for conducting large experiments efficiently, support for parametric values needs to be added. For evaluating individual designs, however, the evaluation is fully automated.

$\mathcal{I}4$ Performance results: Evaluation leads to throughputs, utilizations and response times. DiffLQN does not provide visualizations, e.g., graphs of these results. Validation conveys that analysis based on ODEs computes much faster than computing simulation equivalents. Both ways of analysis display similar results, especially when the thread multiplicities are high, i.e., the estimates are asymptotically exact for large enough degrees of concurrency.

In brief, the high-level model of DiffLQN consists of **LQNs**, which provides a useful abstraction mechanism via **layers**. Moreover, LQNs separate services and resources, in compliance with the **Y-chart** philosophy. DiffLQN provides results via **numerical analysis** and simulation, but does not visualize these results.

B.3 Hierarchical Evaluation Tool

The Hierarchical evaluation tool (HIT, [18, 199]) provides model-based performance evaluation of computing and communication systems during all phases of their life cycle.

$\mathcal{I}1$ High-level performance model: For efficient modelling, HIT provides GUI to design instances. The compact high-level language of HIT explicitly supports the Y-chart philosophy via different abstraction layers of vertical functional hierarchies and horizontal modularization. Model calibration can be done in semi-automatic way by converting measurements into empirical distributions and injecting them into the model.

$\mathcal{I}2$ Underlying performance model: HIT models are transformed into Queueing Network (QN, [10, 70, 86, 92, 114, 196]) on which analysis is performed.

$\mathcal{I}3$ Performance evaluation: Automatic analysis of HIT models is, depending on the subclass of the model, provided by both analytic-algebraical, analytic-numerical, exact and approximate techniques. On top of that, discrete-event simulation (cf. Section 2.2.2) can be applied to all models. DSE can be accomplished by defining variables and their respective ranges.

*I*4 Performance results: Generated results include, among others, populations, throughputs and turn-around-times. For specfic kind of HIT models, e.g., memoryless ones, these results can be obtained not only quickly but also accurately.

In short, HIT provides a **GUI** for modeling in which vertical **hierarchies** and horizontal modules meet, compatible with the **Y-chart** philosophy. Using QNs as the underlying model, various ways of analysis can be applied depending on the model class model. Discrete-event **simulation** can be performed on all models. To the best of our knowledge, HIT does not visualize its results.

B.4 Modeling Language for Reconfigurable Systems

Modeling Language for Reconfigurable Systems (LARES, [74–76,211]) provides a novel approach to the modeling of fault-tolerant systems.

*I*1 High-level performance model: The LARES model [211] describes the structure of a system that is capable of expressing dynamic behavior. It is meant to be convenient and easy-to-learn, and also supports encapsulation, a textual user interface, and a equivalent GUI [74].

*I*2 Underlying performance model: The LARES model can be transformed into various formalisms [75], such as stochastic process algebra (SPA), stochastic petri-nets (cf. Section 2.1.9), and a language for discrete-event simulations. In concrete, a transformation to the SPA tool CASPA has been constructed [6].

*I*3 Performance evaluation: Models with exponentially distributed time delays can be exactly analyzed using Markov chain methods [76]. For the analysis of models with general distributions, however, one has to resort to simulative methods.

*I*4 Performance results: Using the CASPA solver, user defined performance and dependability measures can be obtained. Concretely, the probability of failure and the probability of unsatisfactory performance are often of interest.

To sum things up, LARES provides an convenient and **easy-to-learn** model, which can be transformed in several formalisms. However, for models with general distributions Discrete-event **simulation** is the only way of analysis. A transformation to CASPA and running the CASPA tool, yields a variety of user defined **performance** and dependability measures.

B.5 The Möbius tool

The Möbius tool [41, 43, 93, 190] is particularly used for modeling the behavior of complex systems, such as the reliability, availability, and performance of computer and network systems.

- $\mathcal{I}1$ High-level performance model: Möbius offers either graphical or textual representations. Models are constructed with the right level of detail with respect to the system of interest. The Y-chart philosophy and model calibration are not explicitly supported, but can be modeled manually.
- $\mathcal{I}2$ Underlying performance model: A Möbius model is a hierarchical combination of individual components, such as Stochastic Petri-nets (cf. Section 2.1.9), Markov chains (cf. Section 2.1.7), and stochastic process algebras. Möbius models therefore decompose in a number of low-level models.
- $\mathcal{I}3$ Performance evaluation: Möbius supports (distributed) discrete-event simulation as well as numerical solution techniques. DSE is achieved by combining components in multiple ways.
- $\mathcal{I}4$ Performance results: Exact information about the system, e.g., reliability, availability, performance, and security, can be obtained by conducting measurements at specific time points or periods of time. Numerical solution techniques are exact, but only applicable to a subset of all possible models. Möbius provides ways of data analysis and visualizations.

In short, the Möbius tool offers the **hierarchical construction** of models that consist of, among others, Stochastic Petri-nets, Markov chains, and stochastic process algebras. Evaluation entails **simulation**, but may also be numerical analysis for a subset of models. Additionally, the results encompass measurements, which involve specific points in time or periods of time.

B.6 The Modest toolset

The Modest toolset [83, 87, 88, 144, 214] supports the modeling and analysis of hybrid, real-time, distributed and stochastic systems.

- $\mathcal{I}1$ High-level performance model: The Modest language is a high-level compositional modeling language for stochastic hybrid systems inspired by classical process algebras [83].

*I*2 Underlying performance model: The Modest language can be transformed into a wide array of languages for transition systems (cf. Section 2.1.1 till 2.1.8), including an STA (of Section 2.1.6), PTA (of Section 2.1.5), and DTCM (of Section 2.1.2). One can ensure that a Modest model can be represented as a certain transition system by using a predefined subset of the Modest language.

*I*3 Performance evaluation: A Modest model can be evaluated using the different analysis tools of the Modest toolset. Namely, (i) prohver [67], for evaluating hybrid systems; (ii) MCSTA uses Probabilistic Symbolic Model Checker (PRISM, [128,165]), for probabilistic model checking (only some properties); (iii) MCPTA uses PRISM, for probabilistic model checking; (iv) Model Checking Timed Automata (MCTAU, [24, 83]) uses UPPAAL [19, 132], for traditional model checking; and finally, (v) MODES [83], for discrete-event simulations.

*I*4 Performance results: Evaluations can be used to compute a variety of results, such as response time distributions, throughput, utilizations, and latency breakdown charts. However, the Modest toolset does not visualize these results.

In short, Modest comes with a low-level, **compositional** language that is based on classical **process algebras**. This language is organized into **different classes**, each of them representable as a kind of transition system (cf. Section 2.1.1 till 2.1.8). Evaluation possibilities vary per model class with **simulation** applicable to all classes and model checking to a few classes. The generated results are not visualized.

We would like to point out that iDSL [202–206], the proposed performance language and toolset of this thesis, uses the Modest toolset under the hood. Compared to Modest, iDSL reduces complexity by providing a domain specific language that is more high-level and by delegating each execution to many smaller, low-level Modest executions.

B.7 Modular Performance Analysis and Real-Time calculus

Modular Performance Analysis (MPA, [39, 113, 212]) is an abstraction for the analysis of component-based real-time systems.

$\mathcal{I}1$ High-level performance model: In MPA, UML [65] is used to define distinct application and platform models, adhering to the Y-chart philosophy [50]. Model calibration can be exercised by turning measurements into arrival and services curves.

$\mathcal{I}2$ Underlying performance model: The high-level UML model transforms into real-time calculus [39] that can be graphically depicted.

$\mathcal{I}3$ Performance evaluation: The evaluation of the real-time calculus is delegated to Matlab. Analyzing one scenario takes less than one second, whereas computing 3D trade-off plots in which multiple scenarios are displayed takes in the order of minutes. DSE is supported by defining multiple scenarios.

$\mathcal{I}4$ Performance results: The evaluation yields hard upper and lower bounds, but not necessary tight ones. Graphs can be generated in Matlab [137].

In short, MPA uses **UML** as basis for its approach and supports the **Y-chart** philosophy and **Design Space Eexploration**. The results of MPA, which are hard but not necessary tight upper and lower bounds, can be **visualized** using Matlab.

B.8 The Octopus toolset

The Octopus toolset [13, 14, 192, 194] is centered around the Design Space Exploration Intermediate Representation (DSEIR) language, which is designed to be a language that glues all the components that constitute DSE together. The Octupus toolset comprises domain specific modeling, analysis, search and diagnostic tools.

$\mathcal{I}1$ High-level performance model: DSEIR, the high-level language of the Octopus toolset, contains an application and platform model, in accordance with the Y-chart philosophy [13, 14]. In the application model, the order in which tasks are executed is constrained by a partial ordering on tasks, whereas the platform model is concerned with the scheduling of resources.

$\mathcal{I}2$ Underlying performance model: A DSEIR model can be transformed into, among others, Colored Petri-Nets (CPNs, cf. Section 2.1.9) and Timed Automata (TA, cf. Section 2.1.3).

$\mathcal{I}3$ Performance evaluation: A CPN can be analyzed by CPN Tools (of Appendix B.1) enabling Petri-Net simulation. At the same time, a TA can be model checked using UPPAAL (cf. Appendix B.15). DSE is extensively supported via DSEIR language constructs.

$\mathcal{I}4$ Performance results: Both ways of analysis lead to performance metrics, e.g., throughput and memory usage, and simulation-based traces that can be visualized as Gantt charts [214]. Additionally, the Octupus toolset allows for new underlying performance models and ways of evaluation to be added conveniently.

Briefly, the Octopus toolset revolves around the high-level language DSEIR from which transformations to CPN tools and UPPAAL can be made. This yields traces that are based on **simulation** and visualized using Gantt charts. The DSEIR language supports the **Y-chart** philosophy and Design Space Exporation. The Octopus toolset is designed to be **extensible** in various ways.

B.9 The Palladio framework

The Palladio framework [16, 17, 191] aims to provide a component-based software engineering (CBSE) approach that allows for extra-functional properties, including performance and reliability, to be predicted.

$\mathcal{I}1$ High-level performance model: The Palladio Component Model (PCM, [16, 17, 124, 191]) is close to the software UML [65] and is obtained by augmenting the software with performance specific meta data. The Y-chart philosophy is well presented in the model by processes and resources. Model calibration can be accomplished via arbitrary stochastic distributions functions for specifying component behavior.

$\mathcal{I}2$ Underlying performance model: A transformation from the PCM to Layered Queueing Networks (LQNs, cf. Section 2.1.9) exists [124], as well as to stochastic regular expressions.

$\mathcal{I}3$ Performance evaluation: The analytic solver for PCM instances is based on queuing networks with G/G/1 queues. Moveover, the solver only supports single-user scenarios. On top of that, the simulation based evaluation supports the more generic G/G/n queues and multiple-user scenarios.

*I*4 Performance results: The Palladio framework is specialized in response time analysis. The analytical solver is fast and yields very precise results, while the slower simulations provide more modeling freedom. Aggregated statistical data, such as mean values and standard deviations, are provided using an interface to the statistical package R [105]. Visualizations are rendered using a framework with charting capabilities, which plots both histograms and cumulative distribution functions (CDFs) of predicted response times.

In short, the Palladio framework uses PCM, which is an extended version of UML, as its high-level model. This model represents the **Y-chart** philosophy and model **calibration** well. LQN form the underlying model, enabling analytical solving, albeit limited, and evaluations based on **simulation** to primarily obtain response times. **Visualizations** come in the form of histograms and CDFs.

B.10 Parallel Object-Oriented Specification Language

Parallel Object-oriented Specification Language (POOSL, [59, 69, 209]) is an expressive language to model concurrent hardware/software systems adhering to an Object Oriented (OO) and UML [65] paradigm. POOSL comes with a toolset to analyze its models.

*I*1 High-level performance model: The POOSL language, inspired by process algebras, is a generic language that consists of data types, processes, and clusters, whose behavior is specified using simple process algebra primitives. The POOSL language implicitly supports calibration by turning measurements into Empirical Distribution Functions (EDFs, [5]) that are supported by POOSL. The y-chart philosophy is implicitly supported by incorporating it into the model, e.g., [90]. Recently, POOSL models can be created, edited and debugged using the POOSL IDE [62], which is an Eclipse-plugin [56].

*I*2 Underlying performance model: The POOSL language is implicitly transformed to a Labelled Transition Ssystem (cf. Section 2.1.1), following the probabilistic-nondeterministic Segala model [209]. Rotalumis generates a trace from this transition system on-the-fly without explicitly generating the complete state-space, viz., it is typically too big.

*I*3 Performance evaluation: Simulations can be performed in the highly interactive GUI or at high speed using the high-speed rotalumis simulator [59, 189]. DSE is implicitly supported by adding a control layer to the POOSL language for evaluating many designs. On the downside, support for model checking is lacking, partly because of the expressiveness of the high-level performance model.

*I*4 Performance results: Simulations may lead to, among others, jitter, latency and throughput times. Simulations also return UML-based visualizations, e.g., sequence diagrams in the POOSL IDE, and Gantt [214] charts via a third party tool.

In conclusion, The POOSL language is not explicitly transformed into an underlying performance model. It is a generic language based on **process algebra** and implicitly supports the y-chart philosophy, and model **calibration** via distribution functions. POOSL obtains its results by means of **simulations** only, which can be used obtain to jitter, **latency** and throughput times.

We would like to stress that we used POOSL as the toolset and language for two initial experiments (cf. Section 2.5), as follows. First, we created a transformation from an existing DSL for the Movement Control of iXR systems (cf. Section 2.5.1 and Appendix C). Second, we extended the POOSL library with extra components for Image Processing (cf. Section 2.5.2). Despite the useful results these two experiments provided, exact results are called for based on evaluation techniques, such as model checking (cf. Section 2.2.3) or numerical analysis. This made us decide to switch to Modest as our underlying formalism instead (in Chapter 4).

B.11 Performance Evaluation Process Algebra

Performance Evaluation Process Algebra (PEPA, [28, 99, 151]) comes with an expressive formal language and tool for modelling distributed systems, which provides composition, formality and abstraction.

*I*1 High-level performance model: The PEPA language is a concise process algebra, viz., it only comprises constructs for action, choice, cooperation, hiding and process identification. The PEPA Eclipse Plug-in [152] contains a PEPA editor

$\mathcal{I}2$ Underlying performance model: PEPA models are generally transformed into Ordinary Differential Equations (ODE) and Markov chains (cf. Section 2.1.7).

$\mathcal{I}3$ Performance evaluation: The PEPA Eclipse Plug-in comprises different performance analysers which use Markov chains, ODE methods or simulation. Additionally, PEPA tools provides different ways to analyze models: (i) based on ODE as DiffLQN (see Appendix B.2); (ii) via the the multi-paradigm modelling tool Möbius (cf. Appendix B.5); and, (iii) using the probabilistic model checker PRISM (cf. Appendix B.13).

$\mathcal{I}4$ Performance results: Via ODEs, evaluation quickly yields throughputs, utilizations and response times.

In short, PEPA is **process algebra** with a clear set of constructs. PEPA models can be transformed into ODE and Markov chains, which can in turn be delegated to the tools DiffLQN (of Appendix B.2), Möbius (of Appendix B.5), and PRISM (in Appendix B.13). Throughputs, utilizations and **response times** can be computed based on **ODEs**.

B.12 Platform Independent Petri net Editor

Platform Independent Petri net Editor (PIPE, [45, 49]) is an open source, platform independent Petri-Net tool.

$\mathcal{I}1$ High-level performance model: PIPE uses Petri-Net Markup Language (PNML, [162]) as its high-level language, which can conveniently be modeled using an extensive GUI. The Y-chart philosophy is not explicitly supported in PNML, but can modeled in it.

$\mathcal{I}2$ Underlying performance model: Under the hood, the PNML model transforms into Generalized Stochastic Petri-Nets (cf. Section 2.1.9).

$\mathcal{I}3$ Performance evaluation: Besides simulations, PIPE is compatible with various analysis tools, such as a distributed semi-Markov response time analyzer named SMARTA [48], and a distributed Markovian passage time analyzer HYpergraph-based Distributed Response-time Analyser (HYDRA, [28,48]). HYDRA can explore large state spaces using a parallel algorithm that is tailored to be distributed over a network of commodity PCs.

$\mathcal{I}4$ Performance results: Simulations yield the average numbers per place and mean transition throughputs, using Monte Carlo simulation [143]. SMARTA returns response times, while HYDRA returns passage times.

In conclusion, PIPE uses **Petri-Net** Marking Language as its high-level language, which transforms to **GSPN**. PIPE evaluates models using SMARTA for **response time analysis** and HYDRA for Markovian passage times. Also, simulations return the average numbers per place and mean transition throughputs.

B.13 Probabilistic Symbolic Model Checker

PRISM is a probabilistic model checker, i.e., a tool for formal modelling and analysis of systems that exhibit random or probabilistic behaviour.

$\mathcal{I}1$ High-level performance model: PRISM models are state-based, simple, low-level, and modular. A PRISM model can represent various transition systems.

$\mathcal{I}2$ Underlying performance model: The PRISM language can be used to directly represent discrete-time Markov chains (DTMCs, cf. Section 2.1.2), Continuous-Time Markov Chains (CTMCs), Markov Decision Processes (MDPs, cf. Section 2.1.4), and Probabilistic Timed Automata (PTAs, cf. Section 2.1.5).

$\mathcal{I}3$ Performance evaluation: PRISM models can be simulated using a discrete-event engine, but also automatically evaluated for a wide array of quantitative properties. For instance, the property “what is the probability of a failure causing the system to shut down within 4 hours?" can be represented in PCTL temporal logics [85]. PRISM supports experiments, which is a concept equivalent to checking multiple design instances.

$\mathcal{I}4$ Performance results: By supporting multiple underlying models and temporal logics, PRISM can be used to return a variety of results, of which the quality of service for (probabilistic) communication, network and multimedia protocols [53,130], and the correctness and performance of several security-related systems [147,184].

In short, PRISM models are state-based, simple, low-level and modular. They can be used to represent DTMCs, CTMCs, **MPDs**, PAs and PTAs. Evaluation comprises discrete-event **simulation** and automatic evaluation for quantitative properties, which are expressed in terms of **temporal logics**.

B.14 Software Performance Evaluation

Software Performance Evaluation (SPE, [22, 23]) implements the concepts and steps of the SPE based on the Real time-Unified Modeling Language (RT-UML, [126]) language.

- $\mathcal{I}1$ High-level performance model: SPE uses the RT-UML as its high-level performance language, which is UML [65] extended with performance profiles. RT-UML contains a separate software model (execution graphs) and machine model (queueing networks), in line with the Y-chart philosophy. Additionally, model calibration is possible by creating so-called Schedulability, Performance, and Time (SPT) profiles.

- $\mathcal{I}2$ Underlying performance model: SPE is capable of using, among others, QNs (cf. Section 2.1.9) as its underlying model, which is obtained by combining the the software model and the machine model.

- $\mathcal{I}3$ Performance evaluation: The QN model is evaluated either using either the QN model analytical solver or simulator. DSE is not explicitly supported. Tooling automates all steps from the UML model to the final results, which are conveniently added to the original RT-UML model again.

- $\mathcal{I}4$ Performance results: Evaluation leads to execution times, response times and resource utilizations.

Briefly, SPE is based on a **UML** extended with profiles, i.e., RT-UML, which supports the **Y-chart** philosophy and model calibration via profiles. SPE is used with **QNs** under the hood. Evaluation with an analytic solver or simulator, yields execution times, **response times** and resource utilizations.

B.15 Uppsala Aalborg model checker

Uppsala Aalborg model checker (UPPAAL, [19, 132, 200]) is an integrated tool environment for modeling, validation and verification of real-time systems.

- $\mathcal{I}1$ High-level performance model: To facilitate modeling, UPPAAL provides both graphical and textual formats for the description language. One can use the textual format or the Autograph-based graphical user interface [26] simultaneously.

$\mathcal{I}2$ Underlying performance model: UPPAAL models are networks of timed automata (TA, cf. Section 2.1.3)

$\mathcal{I}3$ Performance evaluation: UPPAAL models can be validated using graphical, interactive simulation, but also be verified via model checking.

$\mathcal{I}4$ Performance results: Model checking leads to reachability, safety, and liveness properties, as well as verifying the absence of deadlocks.

Conclusively, UPPAAL can be used to model **real-time** systems, in both a **graphical** and textual format, as networks of **TA**. UPPAAL models can be validated using graphical simulations and verified via **model checking**. Model checking yields, among other, reachability, safety and liveness properties.

Note that MCTAU, the model checker of the Modest toolset, takes advantage of UPPAAL under the hood [24] for traditional model checking.

APPENDIX C

Performance evaluation for Collision Prevention

This chapter is based on the following publication.

[207] 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 *Lecture Notes in Computer Science*, pages 276–287. Springer, 2013. doi: 10.1007/978-3-642-40725-3_21

C.1 Introduction

Embedded systems are ICT systems designed to operate as part of larger systems and are commonplace in dependable, critical sectors, like aerospace, automotive, health-care and robotics. They have faced a significant complexity increase over time, caused by an ongoing exponential growth of hardware and software. Furthermore, they have faced increased interaction in a dynamic, interactive environment. External interactions of embedded systems are often implemented using control loops consisting of three (indefinitely running) functional steps: sensing (extracting environment information), thinking (from information to decision making) and acting (from decisions into observable behavior).

Control needs to meet performance requirements (throughput and latency) that should be an integral part of the system requirements [9]. Model-based and model-driven design methods have been proposed to improve the complex design process for embedded systems [95, 116, 178], but addressing the performance aspects remains difficult. Particularly, predicting performance in the

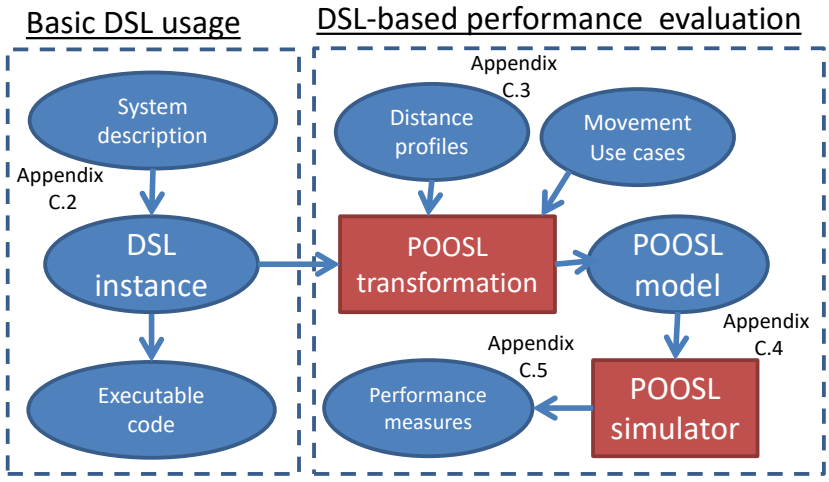


Figure C.1: Basic DSL usage (left), extended with performance evaluation (right)

early design stages is hard, since the system does not exist yet [44, 181]. This appendix addresses early-in-design performance evaluation using a DSL.

We report about our contributions to an industrial study project at Philips Healthcare aimed at redesigning some of the collision prevention (see Section 3.2.1) components, used in their interventional X-ray (iXR, [158, 159]) machines (cf. Chapter 3). Collision prevention strategies vary across product configurations (see Section 3.1.2) and medical applications (see Section 3.1.1). We strive for reuse-ability across product configurations and hence started with specifying the safe functionality using a DSL. In the study, a prototype domain specific language (DSL, [142, 208]) for collision prevention was developed. A DSL instance is a formal system specification, from which executable code is generated; see Figure C.1 (left). Early in the study, distance computations (implemented by a third party package) were identified as performance critical. In the current literature, [31], Proximity Query Package (PQP, [131]) is often mentioned for distance computations. Hence, we have decided to use PQP to illustrate our approach.

In this appendix, we add early (design phase) DSL-performance evaluation to the basic DSL approach, before the embedded system even exists; see Fig-

ure C.1 (right). In addition to the basic approach, the DSL is reused to create a performance model to generate up-to-date performance metrics dynamically. Hence, the performance of DSL-instances can be compared automatically. Formal specification mechanisms that allow reasoning about product families and enable Design Space Exploration (DSE) exist [187, 213]. DSL-performance evaluation requires an automatic transformation from DSL-instances to performance models, applicable to any valid DSL-instance. We used the Parallel Object-oriented Specification Language (cf. Appendix B.10) as modeling language and derive functional-flows of executable functions from DSL-instances. To obtain insight in the execution-times, we added PQP-profiles and use cases as extra model parameters.

Related work For the performance evaluation of embedded software, a variety of techniques has been used recently, both from the more traditional field of Queueing Network (QN, [10, 70, 86, 92, 114, 196]), as from the field of embedded system design. A good overview on techniques for software performance evaluation using a model-based approach is provided in [9]. We address a few recent cases below, without attempting to be exhaustive.

Process algebra models, in particular PEPA (cf. Appendix B.11), have been used for the evaluation of an industrial production cell [99], whereas [28] uses PEPA to specify active badge models to comute so-called passage times. Three different approaches, i.e., timed Petri nets, data flow graphs (SDF) and timed automata (UPPAAL) were compared for the evaluation of an image processing pipeline [103]. The Petri net approach provided the most expressive modelling framework, UPPAAL (cf. Appendix B.15) was most adequate in finding schedules and SDF appeared to be most scalable. [104] describes the evaluation of a printer datapath, where UPPAAL is used to compute worst-case completion times. Using Probabilistic Symbolic Model Checker (PRISM, [128, 165]), a Continuous-Time Markov Chain (CTMC) model of an embedded system is evaluated and shutdown probabilities are obtained. Parallel Object-oriented Specification Language (POOSL, [59, 69, 209]) supports a UML [65] based modelling approach; two recent case studies papers address a production cell model [111] and an in-car navigation system model.

In the current case study, a soft real-time system in which consecutive distance query functions need to be executed is modeled. A worst-case analysis seems less appropriate here, as this assumes the extremely unlikely case that all functionality under-performs coincidently. We therefore turn our attention to discrete-event simulations, supported by the POOSL tool Rotalumis [59], which

provide us with useful statistical results.

Appendix outline This appendix is further organised in five sections that relate to the components of Figure C.1. Section C.2 describes the case study system. Section C.3 specifies the PQP-query profiling, followed by Section C.4 that presents the full DSL-based POOSL model. Section C.5 validates the POOSL performance-model by comparing it to the real system. Section C.6 concludes this appendix.

C.2 System description

iXR systems are used for acquiring patient images using x-ray technology. They consist of large and heavy objects (as illustrated in Figure 3.1) that translate and rotate based on user input. Collision prevention is vital for safety of the patient and implemented by a Movement Control loop. We focus on one collision prevention technique based on 3D-models in which computing the shortest distance between two 3D-objects is the key operation.

C.2.1 The Movement Control loop

The Movement Control loop tracks object-distances frequently. It intervenes in system operation when two objects are too close to each other by overriding user speed-requests by lower ones and demands stable and low response times to ensure timely and reasonable acting. The Movement Control comprises functions that execute using a FIFO scheduling policy in a single-threaded, sequential and non-preemptive manner. At its highest abstraction level, the Movement Control loop is decomposed into three succeeding functions, named `Sense`, `Think` and `Act` (see Figure C.2). The functions are, respectively, responsible for reading geometric sensor positions, reasoning about current geometric positions and decision taking, and sending object speed-requests. `Sense` and `Act` are atomic functions by design. The more complex function `Think` has recently been redesigned using a DSL of which instances are automatically transformed to executable code. We would like to know whether, for a given system configuration and in a variety of scenarios, `Think` executes quickly enough to ensure safety at all times.

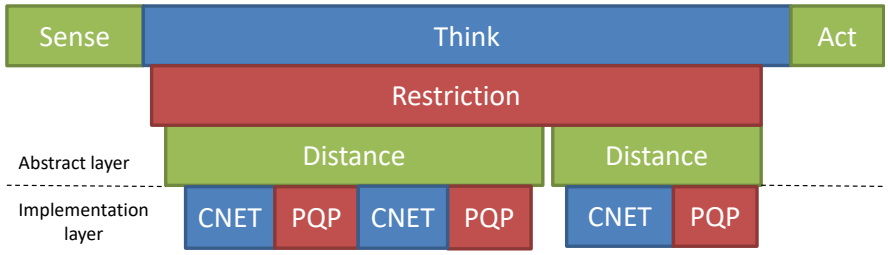


Figure C.2: The hierarchical structure of the Movement Control's functions

C.2.2 Think

`Think` is partitioned several so-called `restrictions`, each constituting of a conditional collision-danger clause and speed statement. The collision-danger clause yields true, if and only if it applies in the currently sensed situation, e.g., a small distance between objects. In this case only, the corresponding speed statement assigns a positive (translational or rotational) speed limit to a geometric object. In more detail, restrictions are implemented by a number of distance queries depending on the DSL-instance at hand. For `Think`, distance queries are the only way to sense, and speed limits the only way to act. Distance queries are performed for two geometrical objects on a specified geometric model.

C.2.3 Distance queries

At a lowest abstraction level, distance queries are executed. For our performance evaluation approach, they have been implemented using the Proximity Query Package (PQP, [131]). Distance queries comprise one or more PQP calls (due to the decomposition of objects), each preceded by one preperatory CNET operation. The CNET operation is a relatively short and constant-time operation, relative to the PQP calls; for that reason, we do not further address it here. PQP provides a 3D-Eucledian distance between two objects, but shows variable execution-times that are hard-to-determine a priori. In the next section, PQP is profiled to gain an understanding of its performance characteristics, so that we can use that information when building the model.

C.3 Profiling the performance of PQP

In this section, a performance profile of PQP is created by measuring its execution times in a variety of representable circumstances, using a 3D model of a real iXR system. We create Empirical Cumulative Distribution Functions (ECDFs) of the execution-times, which are sampled later for simulation purposes. PQP-queries require two sets of triangles as input and execute an heuristic algorithm that selects one triangle per set, such that the distance between triangles is minimal. PQP returns exactly this distance. The key of PQP's performance lies in the way triangles are traversed. PQP uses bounded volumes [131] as an abstraction mechanism for objects. Therefore, PQP is faster than the $O(nm)$ theoretical worst-case (with n and m the number of triangles per object), but this comes at the price of variable execution-times that are hard-to-determine a priori.

Experimental profiling conveyed that both the complexity of the input objects (in Appendix C.3.1) as well as their relative geometric positions (in Appendix C.3.2) affect the performance of PQP significantly. Hence, PQP requires a differentiated way of profiling; there is not just one profile that fits all foreseen queries.

C.3.1 Object complexities

In the following, we deal with object complexities first, after which the relative geometric positions are taken into account. We classify PQP-queries on the basis of input object-pairs to account for run-time variations resulting from different object complexities. In our study, we profiled PQP in isolation from the Movement Control loop in the 3D model. However, as a result we were not able to profile all PQP object-pairs needed for our model. To compensate for this, we estimated the unknown performance of object-pairs using a known measure for object-pair complexity, namely the product of triangles per object [110]. We then matched unknown queries with the nearest, but more difficult query to avoid underestimating profiles.

C.3.2 Relative geometric positions

Additionally, the relative geometric positions of objects influence the execution-times of PQP. This stems from the unpredictable, heuristic way PQP searches for the two distance-defining triangles. PQP operates in a 3-dimensional geometric space in which objects can be translated and/or rotated using 12 dimensions

from their starting position (leading to a total of more than 10^{35} positions). The immense geometric space demands the use of profiling methods that consider a restricted but representative sample of the space of all possible positions, as follows.

We used four profiling methods, named 2D, 3D, 9D grid, and 9D random. The 2D and 3D profiling methods cover geometric positions that match special test cases, which have been designed by Philips to test iXR systems in performance demanding geometric positions. Profiling using few dimensions allows for small step sizes, but is however restricted to a small part of the sample space. In contrast, the 9-dimensional methods profile more distant samples. The 9D grid profiling method varies in 3 of 4 fixed positions per dimensions, whereas the 9D random selects geometric positions with uniform probability. This results in a large part of the search space being covered but comes at the price of potentially missing local maximums.

13467 (2D), 137417 (3D), 88671 (9D-grid) and 90012 (9D-random) geometric positions were sampled. On each of these positions leven PQP distance queries were performed, viz., one per distance pair. PQP-queries were classified for eleven object pairs and four profiling methods, resulting in 44 kinds of PQP-queries. We constructed 44 empirical CDFs for simulation, as follows:

$$\hat{F}_j(t) = \frac{1}{n_j} \sum_{i=1}^{n_j} \mathbf{1}\{x_i^j \leq t\}, \quad j = 1, \cdots, 44,$$

where $\mathbf{1}$ is the usual indicator function, n_j is the sample size, and x_i^j the execution-time of sample i for case j.

While sampling from these CDFs during simulation, we choose to not use interpolation. Instead, we "round-up" to the nearest next higher true sample and return values that have actually been observed during profiling.

In Figure C.3, we show the results for PQP-query number 1, which is the most complex one in terms of object complexities and execution-time, to illustrate the the difference between different profiling methods. As can been seen, the execution times of the 2D and 3D methods are higher than their 9D counterparts. This is a result of using test cases, which are designed to push the machine to its limits and therefore have a bias towards "difficult" geometric positions. In contrast, the 9D methods comprise unbiased sampling and take samples from the whole geometric space, which corresponds to the average case.

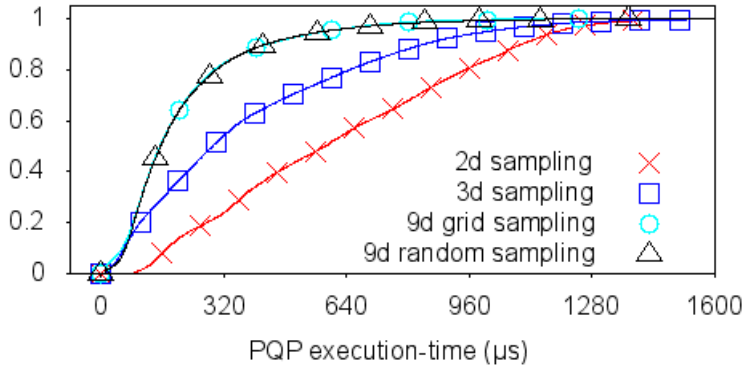


Figure C.3: Profiling PQP_1 using different sampling methods

C.4 POOSL-performance model

In this section, the performance model of the `Think`-component is introduced, which is by far the most time-consuming part of the Movement Control loop. The model is specified in the process algebra and UML-based POOSL language, which is a system-level description and language that enables fast-simulation of the `Think`-component using the Rotalumis engine [59]. POOSL models comprise components that operate autonomously and concurrently, and communicate through lines that connect their interfaces synchronously.

C.4.1 POOSL model outline

In Figure C.4, the constructed POOSL-performance model comprises five components (in red), which require three parameters (in blue). During model creation, we have introduced assumptions to keep the model simple; loops, conditional code executions and functions are assumed independent, which not necessarily reflects reality. Hence, conclusions drawn from the model need to be treated with caution, as illustrated during validation (in Appendix C.5).

The POOSL-model as derived from the DSL corresponds to the `Think`-part of the Movement Control loop, i.e., the sense and act are neglected here. The `Think`-part is triggered by an incoming message at its `start`-port, after which it delegates work, via the components `Cache` and `Distance_query`, to the

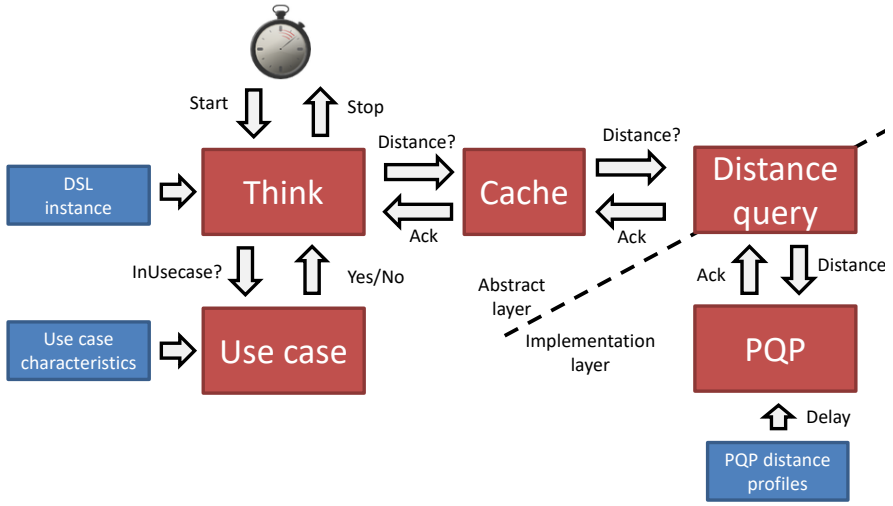


Figure C.4: An overview of the POOSL performance-model of the Movement Control

`PQP` component, by triggering the respective `distance`-ports. In return, messages go all way back through the `ack`-ports to confirm successful executions. Finally, the stopwatch is triggered via the `stop` port, and the elapsed time of one Movement Control loop iteration is stored.

This mechanism yields a single-threaded model, in accordance with the real `Think`-component. The model components have specific responsibilities: `Think` generates distance queries to be performed and forwards them to the `Cache`. `Distance query` decomposes high-level distance queries in one or more low-level distance queries to be performed by PQP. The `Use case` determines which conditional distance queries need execution. The `Cache` filters out redundant distance queries within one Think-loop to ensure intra-loop consistent distance values and enhanced performance. `PQP` simulates PQP-executions by performing delays, based on a profile (of Appendix C.3).

To simulate the `Think`-component, it is required to know which functions execute and how long their executions take. Therefore, the model has been equipped with three parameters: DSL-instances (cf. Appendix C.4.2), use cases (cf. Appendix C.4.3) and PQP profiles (cf. Appendix C.4.4). We explain these

parameters and their role in the performance model.

First, a DSL-instance is transformed into a sequence of (conditional) distance queries that repeats indefinitely, constituting the `Think`-component. For each distance query in `Think`, the `Use case`-component is accessed to decide whether it should be executed or not.

Second, the `Use case`-component is initialized with the `InUseCase` parameter, which comprises a set of use case characteristics. They refer to the conditional distance queries in the `Think`-component and represent the probabilities with which they should execute. Using Monte Carlo sampling [143], the `Use case` maps the probabilities to Boolean values representing either execute or not, which are returned to `Think`.

Third, the PQP component is initialized with execution times that are based on profiles (of Appendix C.3). Distance queries in the `Think`-component that have no cache-hit in `Cache` become PQP-queries in the `PQP`-component. The CDFs of PQP's execution-times are used for sampling execution-times using random numbers. The object pair of the PQP-query determines which CDF is used, while the geometric position and thus the model of the environment are only implicitly considered by the CDF's variation.

C.4.2 The DSL instance

iXR machines need to make safe movements (cf. Section 3.2.1). Therefore, a prototypical DSL with a tailored grammar has been designed that specifies safe object movements by keeping track of object distances and speeds.

In practice, DSL-instances are automatically transformed to executable code (see Figure C.1, left). They declare models and objects that form the basis of distance queries. Additionally, they consist of restrictions that impose a speed limit on an object when an activation condition is met. The condition generally contains one or more distance queries between objects to check whether they are far enough apart. The speed limit enforces a specific object to remain below a speed, potentially overriding higher user requests. The limit can be a constant value, but can also dynamically depend on object distances.

Besides the transformation producing implementation-code, we have constructed a second transformation from DSL-instances to POOSL-model `Think`-components that can be applied to all DSL-instances (see Figure C.1, right). It generates a POOSL `Think`-component with a fixed number of distance queries. However, distance queries (may) execute conditionally for two reasons. First, boolean connectors may be susceptible to lazy evaluation in the executable language. This means that distance queries in their right hand might not execute

depending on the evaluated result of the left hand. Second, the activation may evaluate to false, which causes distance queries in the effect to not be executed. The transformation accounts for this by enclosing conditional POOSL-code fragments with if-statements. We call the conditions of if-statements use-case characteristics. Consequently, the execution of distance-queries varies per loop.

In our case study, we used a DSL-instance with four pairs of restrictions. One pair is listed in Table C.1, which prevent collisions between the TableTop and Beam object. The remaining three pairs are similar, but apply to other object pairs. We explain the first two restrictions. `ApproachingTableTopBeam` (lines 6-11) activates when the objects are within a certain distance (line 8) and approaching each other (line 9). If so, the speed limit is lowered, using a monotone break pattern that maps lower speed limits to lower object distances (line 11). Restriction `CloseTableTopBeam` (lines 5-18) activates for very small distances and yields a speed limit of 0, i.e., an emergency stop (line 18).

```
 1  supervisor
 2
 3  object TableBase, TableTop, Beam, Detector
 4  model Now, Future, NowHyst, FutureHyst
 5
 6  restriction ApproachingTableTopBeam
 7  activation
 8    Distance[FutureHyst](TableTop, Beam) <18 + 125 &&
 9    Distance[Future](TableTop, Beam)
        <Distance[Now](TableTop, Beam) - 0.3
10  effect
11    limit Beam[Rotation] at
        ((Distance[Future](TableTop,Beam) - 25) / 100))
12
13  restriction CloseTableTopBeam
14  activation
15    Distance[NowHyst](TableTop, Beam) <17.5 &&
16    Distance[Future](TableTop, Beam)
        <Distance[Now](TableTop, Beam) + 0.3
17  effect
18    limit TableTop[Translation] at 0
```

Table C.1: A DSL-instance example with 2 restrictions

Restriction 1 ApproachingTableTopBeam	
1	$Distance[FutureHyst]$
2	if (r1a){
3	$Distance[Future]$
4	$Distance_{\lfloor}Now]$
5	if (r1lim) {
6	$Distance[Future]$
7	}
8	}

(a) Restriction 1

Restriction 2 CloseTableTopBeam	
9	$Distance[NowHyst]$
10	if (r2a) {
11	$Distance[Future]$
12	$Distance[Now]$
13	}

(b) Restriction 2

Table C.2: Pseudo code of the POOSL-Think component for two restrictions

In Table C.2, we present pseudo-code of the `Think`-component as automatically derived from the DSL-instance (of Table C.1). It contains seven distance queries, viz., four in restriction 1 (in Table C.2(a)) and three in restriction 2 (in Table C.2(b)). Both restrictions start with an unconditional distance query (pseudo code, lines 1 and 9), since the left operands of the &&-operators in the DSL execute indefinitely (DSL instance, lines 8 and 15). Next, two queries per restriction are conditional (pseudo code, lines 3,4,11 and 12), since the right operands of the &&-operators are evaluated in a lazy way in the execution code (DSL instance, lines 9 and 16). Finally, the first restriction has a fourth distance query (pseudo code line 6) that only executes when both operands of the &&-operators to yield true, because it is located in the activation clause (DSL instance, 11).

C.4.3 Use cases

The system's required functionality depends on the way it is used. Hence, we construct a use-case dependent model. We define a use-case as a set of characteristics, which consist of a label that refers to a fragment of code, and a execution probability each. Table C.3 displays, for four use cases, the probability values for each label (with "other" the default probability for all other labels). A label is decomposed into a fixed prefix "r", restriction number (1 to 8) and restriction part (activation or limit) in line with the pseudo code (cf. Table C.2).

The corresponding value represents the probability that the corresponding fragment of conditional code gets executed. For instance, the value 0.29 of r2a

Label	r2a	r2lim	r3a	r4a	r4lim	other
UC1	0.29	0.99	0.02	0.11	0.71	0
UC2	0.07	0.04	0	0	0	0
UC3	0.01	0.79	0.01	0.12	0.62	0
UCΩ	1	1	1	1	1	1

Table C.3: For use case 1,2,3 and Ω, characteristics (label and probability value)

for use case 1, indicates that in restriction 2 distance query 2 and 3 (pseudo code, lines 11 and 12) get executed in 29% of the cases. This is implemented in the POOSL-model by performing two-way communication. First, `Think` provides a label. After this, `Use case` looks up the probability value and turns it into a Boolean value by comparing it to a random number obtained via Monte Carlo sampling [143]. This Boolean value is returned.

C.4.4 PQP profiles

In Section C.3, the creation of PQP-profiles to enable the sampling of execution times for simulation was discussed. For this purpose, the profiles are injected into the `PQP`-component of the POOSL-model, which simulates the execution of a CNET-operation, for converting object to triangles, followed by a PQP-query based on a distance pair. The execution of CNET took 517 μs on average with a 58 μs standard deviation. We model this as an uniform distribution U(459,575) distant-pair invariant. The `PQP`-component receives distance queries (with an object pair parameter) from `Think`. `PQP` selects the right CDF (belonging to the object pair), draws a sample from this CDF and simulates a CNET-operation followed by a PQP-query (using a POOSL delay). An acknowledgement is then sent back to `Distance Query`. We have contributed to the POOSL-language by constructing a new distribution subclass for empirical CDFs.

C.5 Validating the movement-control model

In this section, we asses the validity of the movement-control performance model and refer to `Think` as `Think`-prototype, as it is based on a DSL-prototype. This section is organized, as follows. In Appendix C.5.1, the experimental

set-up is provided, after which we introduce the different use cases (in Appendix C.5.2, C.5.3, C.5.4 and C.5.5). Finally, the outcomes of the real software and the performance model are compared using the Kolmogorov distance (in Appendix C.5.6) and Execution time ratio (in Appendix C.5.7) measure.

C.5.1 Experimental set-up

To reach an experimental set-up, we have selected values for the three parameters of the POOSL-model, as follows:

1 DSL instance: We have used the shown DSL-instance (of Table C.1) for all experiments.

2 Use case: We have used four different use cases (cf. Table C.3). Use cases can be seen as a sequence of geometric positions representing dynamic machine positions. These positions affect the performance of PQP-queries by affecting the relative object positions, an effect neglected by the model. Additionally, the model does take the variable execution of functionality into account. For this purpose, use cases are mapped to a numbers of characteristics (see Tabel C.3), which each represent the execution probability of a certain code fragment.

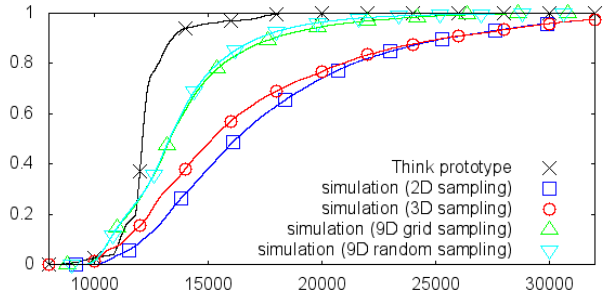
3 PQP profiling: We have used simulations based on four different PQP profiling methods, which are 2D, 3D, 9D-grip and 9D-random (as discussed in Section C.3).

We have performed experiments for three use cases (cf. Figure **??**), as follows. We performed a real-time execution on `Think`-prototype machine by providing (use case-driven) user inputs, for 278, 284 and 301 seconds. Next, we performed four simulations (one per profiling method) for three use cases, using the use case characteristics from the corresponding execution.

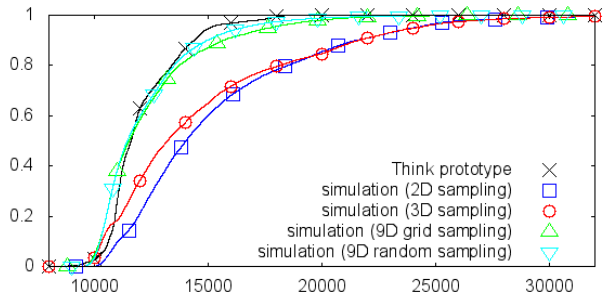
Finally, we plotted the distributions of the four simulations and execution in one graph per use case. The experiments were performed on one PC (i5-2400, QuadCPU@3.10Ghzm 3Gb RAM) to execute `Think`-prototype and another PC (AMD A6-3400m, QuadCPU@1.4Ghz, 6Gb RAM) to simulate the POOSL-model.

C.5.2 Use case 1: Arc movements

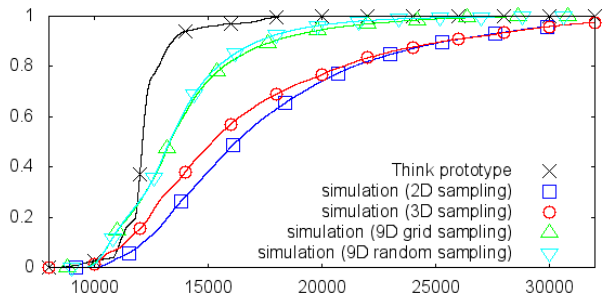
The `Arc`-object has been rotated in various ways around and towards the `Table`-object, while the `Table` remained motionless. iXR systems make pictures from



(a) use case 1

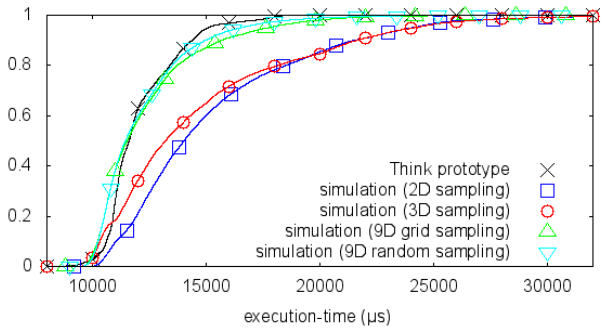


(b) use case 2



(c) use case 3

Figure C.5: The execution-time CDFs for four use cases, with the cumulative probability (y-axis) for an execution time (x-axis).



(d) use case Ω

Figure C.5: The execution-time CDFs for four use cases, with the cumulative probability (y-axis) for an execution time (x-axis).

various angles this way. Figure C.5(a) displays the experiment outcomes. Simulations based on 2D and 3D profiling show pessimistic results, whereas the ones based on 9D-profiling match `Think`-prototype better. We attribute the difference to the most complex PQP_1 query (of Figure C.3) that conveys an overestimation for 2D and 3D-based profiling. Underestimations occur for 9D-simulations, but only for low execution-times.

C.5.3 Use case 2: Table movements

The `Table`-object has been moved towards and away from the `Arc` mimicking situations in which patients enter and leave iXR systems. The `Arc` moved every now and then too, to avoid repetition. Figure C.5(b) shows that 9D-simulations match the `Think`-prototype execution well. Therefore, replacing unknown PQP-queries with similar ones and adding independences to the model did not affect the results much. Simulations based on 2D and 3D-profiling methods overestimate execution-times, which we again contribute to the PQP_1 profile. Again, 9D-simulations underestimate for low execution-times.

C.5.4 Use case 3: Stationary objects

Stationary behaviour that is common for iXR machines, has been performed in this use case. Nevertheless, computations took place due to the particular DSL-instance, particularly when objects are close to each other. All stationary

positions had geometric positions that resemble those in previous use cases. Figure C.5(c) displays the results for use case three. The curve of the `Think`-prototype contains three points of infliction. They are presumably caused by spending time in two clusters of geometrical positions and yield execution-times of around 12 and 16 milliseconds, respectively. On average, the simulation outcomes match the `Think`-prototype well, but is not able to deal with the points of inflictions which require use case characteristics to be dependent. In line with the previous two use cases, simulations based on 2D and 3D profiles appear again more pessimistic than their 9D-counterparts. In this case, this makes them match the `Think`-prototype well for the most part. However, they convey optimistic results around the points of infliction. Therefore, point of inflictions are a phenomenon that requires more attention.

C.5.5 Use case Ω: Universal machine

This use case uses 1-values for all its characteristics, so that all conditional code is always executed. Consequently, machines that support use case Ω, can handle all conceivable use cases including the previous three. Use case Ω has not been executed on the `Think`-prototype. Figure C.5(d) (right-bottom) shows the simulation results for use case Ω with the highest execution-times as a result of executing more code.

We draw a two-fold conclusion from the experiments. First, applying use cases saves dramatically on hardware resources and determining the use case characteristics is a delicate procedure that can lead to the underestimation of execution-times. Second, the execution-time ratio between use case Ω and others is DSL-dependant, viz., it is determined by the proportional amount of conditional code.

C.5.6 Kolmogorov distances

We computed Kolmogorov distances (a CDF distance measure) between the `Think`-prototype and corresponding simulations to compare their results. The lower the distance, the better the simulation performed. Table C.4 shows the distances between the simulations and the `Think` prototype, per use case and profiling method, computed as follows:

$$K = \sup_x |F(x) - G(x)|, \tag{C.1}$$

	2D	3D	9D grid	9D random
UC1	0.67	0.58	0.41	0.41
UC2	0.42	0.30	0.16	0.16
UC3	0.29	0.18	0.27	0.30

Table C.4: Kolmogorov distances between simulations and Think-prototype, per use case and profiling method

	2D	3D	9D grid	9D random
UC1	2.02	2.03	1.53	1.68
UC2	1.79	1.79	1.54	1.72
UC3	1.81	1.81	1.55	1.58

Table C.5: Maximum execution-time ratio between simulations and Think-prototype, per use case and profiling method.

where K is the distance, and F and G the CDFs that are compared.

Generally, the higher the number of dimensions the profiling method covers, the lower the distances are. Both 9D-simulations show Kolmogorov distances that match each other, which is confirmed by them having similar CDFs for all use cases. Since a large amount of sampling points has been used, we conclude that for the current geometrical domain it does not matter whether the grid or random method is used. `Think`-prototype conveyed an irregular CDF with infliction points. Although the model was not able to reproduce this pattern, the Kolmogorov distances turned out low. Additionally, we determined how quick simulations converged, which means that simulating longer does not change the results much. We computed the Kolmogorov distance between short simulations runs (0.25, 1, 15 and 75 seconds) and the longer runs of the experiments. 2D and 3D-simulations converged within a second, while the 9D-ones needed 15 seconds.

C.5.7 Execution-time ratios

Finally, we have compared the ratios of execution-times between the simulations and the `Think`-prototype (Figure C.5) as an alternative measure to the

Kolmogorov distance. Namely, we picked the maximum ratio for each CDF, computed as follows:

$$R = sup\left\{\frac{x}{y} \mid F(x) = G(y) \wedge y > 0\right\}, \tag{C.2}$$

where R is the execution-time ratio, and F and G the CDFs that are compared.

2D and 3D simulations show the biggest ratios greater than 2, which were recorded for high-execution times. On the other hand, lower ratios occurred for lower execution-times. This means that the simulations execution-times have a higher variance than `Think`-prototype.

Discussion The validity of the model has been evaluated by comparing POOSL-simulations with `Think`-prototype executions. Executions led to three use cases with their characteristics, which were used as simulation inputs. We address the concerns the reuse of execution information might raise, but have been very open about what these use cases are.

Simulations yielded higher execution-times than the `Think`-prototype, resulting from the profiling methods (both the sample space selection and overestimation of unknown queries). On top of this, 2D and 3D sampling showed higher execution-times then 9D-ones, as a result of test case-based profiling. Overestimation is wanted to some extend, since underestimation might lead to non-safe machines in the very end. However, over-dimensioned hardware literally has its price.

Table C.5 shows the execution-time ratios. 2D and 3D-simulations lead to ratios greater than 2, while the 9D-ones remained below 1.6. Finally, simulations of various lengths have been performed and conveyed that the simulations performed were of sufficient length.

C.6 Conclusion

Embedded systems have increased in complexity over time (see also Section 1.1), making early-in-design performance evaluation indispensable (cf. Section 1.2). For this purpose, we have constructed a DSL-based POOSL-performance model for `Think`-prototype of iXR medical systems. Addtionally, we profiled distance queries and introduced use cases to make the model more situation specific. In

our model, we presumed distance queries, execution of conditional code fragments and subsequent loops independent. We evaluated the validity of the model by comparing POOSL-simulations with `Think`-prototype executions. In general, the model overestimated the execution-times of `Think`-prototype with a ratio of 1.6 overestimation. Simulations converged within 15 seconds, enabling quick feedback.

We compare the approach in which a transformation from DSL to a performance model (e.g. POOSL) is used with one in which the performance model is generated manually. The former approach separates the roles of the domain expert and performance analysis expert, allowing both parties to solely focus on their areas of concern. Additionally, the DSL-transformation makes it possible to generate multiple performance model instances, e.g., to support different product families or compare design alternatives, based on different DSL-instances. A manual low-level approach does not restrict the definition of similar components in a variety of ways, making high-level changes laborsome. DSL-approaches counteract this by defining high-level concepts by default. Compared to a manual performance model, DSLs make the switch to other performance techniques easier. Similarly, a DSL-instance can, using multiple transformations, be the source of different artifacts, such as code, models and documentation.

Bibliography

[1] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. John Wiley and Sons, 1995. ISBN: 978-0-471-93059-4.

[2] 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. doi: 10.1109/msp.2013.49.

[3] R. Ammar, M. Farid, and K. Yetongnon. A spreadsheet performance approach to integrate a modeling hierarchy of software systems. In *Systems, Man and Cybernetics*, pages 847–852. IEEE, 1989. doi: 10.1109/icsmc.1989.71414.

[4] AnyLogic. AnyLogic: Multimethod Simulation Software. `http://www.anylogic.com/`. Last accessed 2016-12-09.

[5] M. Ayer, D. Brunk, G. Ewing, W. Reid, and E. Silverman. An empirical distribution function for sampling with incomplete information. *The Annals of Mathematical Statistics*, 26(4):641–647, 1955. doi: 10.1214/aoms/1177728423.

[6] J. Bachmann, M. Riedl, J. Schuster, and M. Siegle. An Efficient Symbolic Elimination Algorithm for the Stochastic Process Algebra Tool CASPA. *Software Seminar: Theory and Practice of Computer Science*, pages 485–496, 2009. doi: 10.1007/978-3-540-95891-8_44.

[7] C. Baier, B.R. Haverkort, H. Hermanns, and J.-P. Katoen. Model-checking algorithms for continuous-time Markov chains. *IEEE Transactions on Software Engineering*, 29(6):524–541, 2003. doi: 10.1109/tse.2003.1205180.

[8] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *IEEE Computer*, 36(4):45–52, 2003. doi: 10.1109/mc.2003.1193228.

[9] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-Based Performance Prediction in Software Development: A Survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004. doi: 10.1109/tse.2004.9.

[10] S. Balsamo and G. Iazeolla. Product-Form Synthesis of Queueing Networks. *IEEE Transactions on Software Engineering*, 11(2):194–199, 1985. doi: 10.1109/tse.1985.232194.

[11] J. Banks, J. Carson, B. Nelson, and D. Nicol. *Discrete-Event System Simulation (3rd Edition)*. Prentice Hall, 2000. ISBN: 978-0-130-88702-3.

[12] L. Barroso and U. Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40(12):33–37, 2007. doi: 10.1109/mc.2007.443.

[13] A. Basten, M. Hendrix, N. Trcka, L. Somers, M. Geilen, Y. Yang, H. Corporaal, G. Igna, F. Vaandrager, S. Smet, M. Voorhoeve, and W. van der Aalst. Model-Driven Design-Space Exploration for Software-Intensive Embedded Systems. In *Model-Based Design of Adaptive Embedded Systems*, pages 189–244. Springer, 2013. doi: 10.1007/978-1-4614-4821-1_7.

[14] A. 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 *Lecture Notes in Computer Science*, pages 90–105. Springer, 2010. doi: 10.1007/978-3-642-16558-0_10.

[15] D. Beauquier. On probabilistic timed automata. *Theoretical Computer Science*, 292(1):65–84, 2003. doi: 10.1016/s0304-3975(01)00215-8.

[16] S. Becker, H. Koziolek, and R. Reussner. Model-Based performance prediction with the Palladio Component Model. In *Proceedings of the 6th international workshop on Software and performance*, pages 54–65. ACM, 2007. doi: 10.1145/1216993.1217006.

[17] S. Becker, H. Koziolek, and R. Reussner. The Palladio Component Model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009. doi: 10.1016/j.jss.2008.03.066.

[18] H. Beilner, J. Mäter, 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. doi: 10.1007/978-1-4613-0533-0_5.

[19] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems. In *Hybrid Systems*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer, 1995. doi: 10.1007/bfb0020949.

[20] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*, pages 87–124. Springer, 2004. doi: 10.1007/978-3-540-27755-2_3.

[21] L. Benini, A. Bogliolo, and G. D. Micheli. A Survey of Design Techniques for System Level Dynamic Power Management. *IEEE Transactions On Very Large Scale Integration (VLSI) Systems*, 8(3):299–316, June 2000. doi: 10.1109/92.845896.

[22] A. Bertolino, E. Marchetti, and R. Mirandola. Real-time UML-based performance engineering to aid manager's decisions in multi-project planning. In *ACM Workshop on Software and Performance*, pages 251–261, 2002. doi: 10.1145/584408.584410.

[23] A. Bertolino and R. Mirandola. Towards Component Based Software Performance Engineering. In 25^{th} *International Conference on Software Engineering*, pages 1–6. ACM/IEEE, 2003.

[24] J. Bogdoll, A. David, A. Hartmanns, and H. Hermanns. MCTAU: Bridging the Gap between Modest and UPPAAL. In *Proceedings* 19^{th} *International SPIN Workshop on Model Checking of Software*, volume 7385 of *Lecture Notes in Computer Science*, pages 227–233. Springer, 2012. doi: 10.1007/978-3-642-31759-0_16.

[25] A. Borshchev, Y. Karpov, and V. Kharitonov. Distributed simulation of hybrid systems with AnyLogic and HLA. *Future Generation Computer Systems*, 18(6):829–839, 2002. doi: 10.1016/s0167-739x(02)00055-9.

[26] A. Bouali, A. Ressouche, V. Roy, and R. de Simone. The FC2TOOLS Set. In *CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 441–445. Springer, 1996. doi: 10.1007/3-540-61474-5_98.

[27] J.-Y. L. Boudec and P. Thiran. *Network Calculus: a theory of deterministic queuing systems for the internet*. Springer Berlin Heidelberg, 2001. ISBN: 978-354-045-318-5.

[28] J. Bradley, N. Dingle, S. Gilmore, and W. Knottenbelt. Extracting passage times from PEPA models with the HYDRA tool: A case study. In *Proceedings of the* 19^{th} *Annual UK Performance Engineering Workshop*, pages 79–90, 2003. doi: 10.1109/mascot.2003.1240679.

[29] T. Brázdil, K. Chatterjee, M. Chmelik, V. Forejt, J. Kretínský, M. Z. Kwiatkowska, D. Parker, and M. Ujma. Verification of Markov Decision Processes Using Learning Algorithms. In *Automated Technology for Verification and Analysis*, volume 8837 of *Lecture Notes in Computer Science*, pages 98–114. Springer, 2014. doi: 10.1007/978-3-319-11936-6_8.

[30] R. Calheiros, R. Ranjan, A. Beloglazov, C. De Rose, and R. Buyya. CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011. doi: 10.1002/spe.995.

[31] S. Carpin, C. Mirolo, and E. Pagello. A performance comparison of three algorithms for proximity queries relative to convex polyhedra. In *Proceedings of International Conference on Robotics and Automation*, pages 3023–3028. IEEE, 2006. doi: 10.1109/robot.2006.1642161.

[32] C. Cassandras and S. Lafortune. Stochastic Timed Automata. In *Introduction to Discrete Event Systems*, volume 11 of *The Kluwer International Series on Discrete*

Event Dynamic Systems, pages 317–365. Springer, 1999. doi: 10.1007/978-1-4757-4070-7_6.

[33] Y. Censor. Pareto optimality in multiobjective problems. *Applied Mathematics and Optimization*, 4(1):41–59, 1977. doi: 10.1007/bf01442131.

[34] T. Chen, M. Diciolla, M. Kwiatkowska, and A. Mereacre. Quantitative verification of implantable cardiac pacemakers. In *Proceedings 33rd Real-Time Systems Symposium*, pages 263–272. IEEE, 2012. doi: 10.1109/rtss.2012.77.

[35] CIT. Center for Information Technology. `http://www.rug.nl/society-business/centre-for-information-technology/`. Last accessed 2016-12-09.

[36] E. Clarke. Model Checking – My 27-Year Quest to Overcome the State Explosion Problem. In *Logic for Programming, Artificial Intelligence, and Reasoning*, page 182. Springer, 2008. doi: 10.1007/978-3-540-89439-1_13.

[37] E. Clarke, W. Klieber, M. Novácek, and P. Zuliani. Model Checking and the State Explosion Problem. In *LASER Summer School*, volume 7682 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 2011. doi: 10.1007/978-3-642-35746-6_1.

[38] K. Clohessy, B. Barry, and P. Tanner. New Complexities in the Embedded World - The OTI Approach. In *European Conference on Object-Oriented Programming*, volume 1357 of *Lecture Notes in Computer Science*, pages 472–481. Springer, 1997. doi: 10.1007/3-540-69687-3_87.

[39] Computer Engineering and Networks Laboratory (TIK), ETH. Modular Performance Analysis with Real-Time Calculus. `http://www.mpa.ethz.ch/`. Last accessed 2016-12-09.

[40] R. Conway. Some tactical problems in digital simulation. *Management Science*, 10(1):47–61, 1963. doi: 10.1287/mnsc.10.1.47.

[41] T. Courtney, S. Gaonkar, K. Keefe, E. Rozier, and W. Sanders. Möbius 2.3: An extensible tool for dependability, security, and performance evaluation of large and complex system models. In *IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 353–358. IEEE Computer Society, 2009. doi: 10.1109/dsn.2009.5270318.

[42] CPN Group. Coloured Petri-net tools. `http://cpntools.org/`. Last accessed 2016-12-09.

[43] D. Daly, D. Deavours, J. Doyle, P. Webster, and W. Sanders. Möbius: An Extensible Tool for Performance and Dependability Modeling. In *Computer Performance Evaluation / TOOLS*, volume 1786 of *Lecture Notes in Computer Science*, pages 332–336. Springer, 2000. doi: 10.1007/3-540-46429-8_25.

[44] T. de Gooijer, A. Jansen, H. Koziolek, and A. Koziolek. An industrial case study of performance and cost design space exploration. In *Proceedings of the 3rd International Conference on Performance Engineering*, pages 205–216. WOSP/SIPEW, ACM, 2012. doi: 10.1145/2188286.2188319.

[45] Department of Computing, Imperial College London. Platform Independent Petri net Editor. `http://pipe2.sourceforge.net/`. Last accessed 2016-12-09.

[46] L. Devroye. Sample-based non-uniform random variate generation. In *Proceedings of the 18th Winter simulation conference*, pages 260–265. ACM, 1986. doi: 10.1145/318242.318443.

[47] DiffLQN. Differential Equation Analysis of Layered Queuing Networks - homepage. `http://sysma.imtlucca.it/tools/difflqn/`. Last accessed 2016-12-09.

[48] N. Dingle, P. Harrison, and W. Knottenbelt. Uniformization and hypergraph partitioning for the distributed computation of response time densities in very large Markov models. *Journal of Parallel and Distributed Computing*, 64(8):908–920, 2004. doi: 10.1016/j.jpdc.2004.03.017.

[49] N. Dingle, W. Knottenbelt, and T. Suto. PIPE2: a tool for the performance evaluation of generalised stochastic Petri Nets. *SIGMETRICS Performance Evaluation Review*, 36(4):34–39, 2009. doi: 10.1145/1530873.1530881.

[50] F. Dittmann, A. Rettberg, and F. Schulte. A Y-Chart Based Tool for Reconfigurable System Design. In *Workshop on Dynamically Reconfigurable Systems*, pages 67–73. VDE Verlag, 2005.

[51] K. Doi. Diagnostic imaging over the last 50 years: research and development in medical imaging science and technology. *Physics in medicine and biology*, 51(13), 2006. doi: 10.1088/0031-9155/51/13/r02.

[52] H. Dou, Y. Qi, W. Wei, and H. Song. A two-time-scale load balancing framework for minimizing electricity bills of Internet Data Centers. *Personal and Ubiquitous Computing*, 20:681–693, 2016. doi: 10.1007/s00779-016-0941-9.

[53] M. Duflot, M. Z. Kwiatkowska, G. Norman, and D. Parker. A formal analysis of bluetooth device discovery. *International Journal on Software Tools for Technology Transfer*, 8(6):621–632, 2006. doi: 10.1007/s10009-006-0014-x.

[54] C. Ebert and C. Jones. Embedded Software: Facts, Figures, and Future. *IEEE Computer*, 42(4):42–52, 2009. doi: 10.1109/mc.2009.118.

[55] Eclipse. Eclipse - The Eclipse Foundation open source community website. `http://www.eclipse.org`. Last accessed 2016-12-09.

[56] Eclipse. Eclipse IDE for Java and DSL Developers. `http://www.eclipse.org/downloads/packages/eclipse-ide-java-and-dsl-developers/junosr2`. Last accessed 2016-12-09.

[57] Eclipse. Xtend - Modernized Java - Eclipse. https://eclipse.org/xtend.

[58] Eclipse. Xtext - Language Engineering Made Easy! https://eclipse.org/Xtext/. Last accessed 2016-12-09.

[59] Eindhoven University of Technology. Software/Hardware Engineering - High-Speed Simulation of POOSL Models with Rotalumis. hhttp://poosl.esi.nl/download/. Last accessed 2016-12-09.

[60] C. Eisentraut, H. Hermanns, and L. Zhang. On Probabilistic Automata in Continuous Time. In 25th *Annual IEEE Symposium on Logic in Computer Science*, pages 342–351. IEEE Computer Society, 2010. doi: 10.1109/lics.2010.41.

[61] J. Ellson, E. Gansner, L. Koutsofios, S. North, and G. Woodhull. Graphviz—open source graph drawing tools. In *Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, pages 483–484. Springer, 2002. doi: 10.1007/3-540-45848-4_57.

[62] Embedded Systems Innovation by TNO. POOSL-IDE: Parallel object-oriented specification language. http://poosl.esi.nl/. Last accessed 2016-12-09.

[63] G. Fishman. *Concepts and methods in discrete event digital simulation*. John Wiley, 1973.

[64] C. Forbes, M. Evans, N. Hastings, and B. Peacock. *Emperical Distribution Function*, pages 79–83. John Wiley & Sons, Inc., 2010. doi: 10.1002/9780470627242.

[65] M. Fowler. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004. ISBN: 978-032-119-368-1.

[66] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi. Enhanced modeling and solution of layered queueing networks. *IEEE Transactions on Software Engineering*, 35(2):148–161, 2009. doi: 10.1109/tse.2008.74.

[67] M. Fränzle, E. Hahn, H. Hermanns, N. Wolovick, and L. Zhang. Measurability and safety verification for stochastic hybrid systems. In *Proceedings of the* 14th *International Conference on Hybrid Systems: Computation and Control*, pages 43–52. ACM, 2011. doi: 10.1145/1967701.1967710.

[68] A. Gandhi, M. Harchol-Balter, and M. Kozuch. Are sleep states effective in data centers? In *Proceedings of International Green Computing Conference*, pages 1–10. IEEE, 2012. doi: 10.1109/igcc.2012.6322260.

[69] M. Geilen, J. Voeten, P. Van Der Putten, L. J. van Bokhoven, and M. Stevens. Object-oriented modelling and specification using she. *Computer Languages*, 27(1):19–38, 2001. doi: 10.1016/s0096-0551(01)00014-5.

[70] E. Gelenbe and G. Pujolle. *Introduction to Queueing Networks*. John Wiley and Sons, Chichester, 1998. ISBN: 978-0-471-96294-6.

[71] R. Ghodsi, M. Skandari, M. Allahverdiloo, and S. H. Iranmanesh. A new practical model to trade-off time, cost, and quality of a project. *Australian Journal of Basic and Applied Sciences*, 3(4):3741–3756, 2009.

[72] C. Girault and R. Valk. *Petri Nets for Systems Engineering: A Guide to Modeling, Verification, and Applications*. Springer-Verlag, Berlin, 2003. ISBN: 978-3-540-41217-5.

[73] GNUplot. gnuplot homepage. `http://www.gnuplot.info/`. Last accessed 2016-12-09.

[74] A. Gouberman, C. Grand, M. Riedl, and M. Siegle. An IDE for the LARES Toolset. In *Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance*, pages 240–254. Springer, 2014. doi: 10.1007/978-3-319-05359-2_17.

[75] A. Gouberman, M. Riedl, J. Schuster, and M. Siegle. A modelling and analysis environment for LARES. In *International Conference on Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance*, pages 244–248. Springer, 2012. doi: 978-3-642-28540-0_19.

[76] A. Gouberman, M. Riedl, and M. Siegle. Transformation of LARES performability models to continuous-time Markov reward models. In *Proceedings of 7th International Workshop on Verification and Evaluation of Computer and Communication Systems*, 2013.

[77] GraphViz. Graphviz - Graph Visualization Software. `http://www.graphviz.org/`. Last accessed 2016-12-09.

[78] R. Gronback. *Eclipse modeling project: a domain-specific language (DSL) toolkit*. Pearson Education, 2009. ISBN: 978-364-200-109-3.

[79] M. Grottke, V. Apte, K. Trivedi, and S. Woolet. Response time distributions in networks of queues. In *International Series in Operations Research & Management Science*, pages 587–641. Springer, 2011. doi: 10.1007/978-1-4419-6472-4_14.

[80] E. Grube, G. Schuler, L. Buellesfeld, U. Gerckens, A. Linke, P. Wenaweser, B. Sauren, F.-W. Mohr, T. Walther, B. Zickmann, et al. Percutaneous aortic valve replacement for severe aortic stenosis in high-risk patients using the second-and current third-generation self-expanding corevalve prosthesis: device success and 30-day clinical outcome. *Journal of the American College of Cardiology*, 50(1):69–76, 2007. doi: 10.1016/j.jacc.2007.04.047.

[81] D. Guck, H. Hatefi, H. Hermanns, J.-P. Katoen, and M. Timmer. Modelling, Reduction and Analysis of Markov Automata. *Computing Research Repository*, abs/1305.7050, 2013. doi: 10.1007/978-3-642-40196-1_5.

[82] E. Hahn, A. Hartmanns, and H. Hermanns. Reachability and Reward Checking for Stochastic Timed Automata. *Electronic Communications of European Association of Software Science and Technology*, 70, 2014.

[83] 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. doi: 10.1007/s10703-012-0167-z.

[84] E. M. Hahn and A. Hartmanns. A Comparison of Time- and Reward-Bounded Probabilistic Model Checking Techniques. In *Dependable Software Engineering: Theories, Tools, and Applications*, volume 9984 of *Lecture Notes in Computer Science*, pages 85–100, 2016. doi: 10.1007/978-3-319-47677-3_6.

[85] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal aspects of computing*, 6(5):512–535, 1994. doi: 10.1007/bf01211866.

[86] P. Harrison. Response Time Distributions in Queueing Network Models. In *Performance/SIGMETRICS Tutorials*, volume 729 of *Lecture Notes in Computer Science*, pages 147–164. Springer, 1993. doi: 10.1007/bfb0013852.

[87] A. Hartmanns. Model-Checking and Simulation for Stochastic Timed Systems. In *Proceedings 9th International Symposium on Formal Methods for Components and Objects*, volume 6957 of *Lecture Notes in Computer Science*, pages 372–391. Springer, 2010. doi: 10.1007/978-3-642-25271-6_20.

[88] 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. doi: 10.1007/978-3-642-54862-8_51.

[89] A. Hartmanns and H. Hermanns. In the Quantitative Automata Zoo. *Science of Computer Programming*, 112(part 1):3–23, 2015. doi: 10.1016/j.scico.2015.08.009.

[90] S. Haveman, G. Bonnema, and F. van den Berg. Early insight in systems design through modeling and simulation. *Procedia Computer Science*, 28:171–178, 2014. doi: 10.1016/j.procs.2014.03.022.

[91] S. Haveman and M. Bonnema. Requirements for High Level Models Supporting Design Space Exploration in Model-based Systems Engineering. In *Procedia Computer Science*, volume 16 of *Procedia Computer Science*, pages 293–302. Elsevier, 2013. doi: 10.1016/j.procs.2013.01.031.

[92] B.R. Haverkort. *Performance of computer communication systems - a model-based approach*. Wiley, 1998. ISBN: 978-0-471-97228-0.

[93] B.R. Haverkort and R. Harper. Performance and dependability techniques and tools. *Performance Evaluation*, 44(1-4):1–4, 2001. doi: 10.1016/s0166-5316(00)00067-5.

[94] T. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, Logic in Computer Science '96, page 278. IEEE Computer Society, 1996. doi: 10.1109/lics.1996.561342.

[95] T. Henzinger and J. Sifakis. The Embedded Systems Design Challenge. In *Formal Methods*, volume 4085, pages 1–15. Springer, 2006. doi: doi: 10.1007/11813040_1.

[96] H. Hermanns and J.-P. Katoen. The How and Why of Interactive Markov Chains. In *International Symposium on Formal Methods for Components and Objects*, volume 6286 of *Lecture Notes in Computer Science*, pages 311–337. Springer, 2009. doi: 10.1007/978-3-642-17071-3_16.

[97] S. Hettinga. Performance Analysis for Embedded Software Design. *Master's thesis, University of Twente*, 2010.

[98] J. Hoffmann. *A Heuristic for Domain Independent Planning and Its Use in an Enforced Hill-Climbing Algorithm*, pages 216–227. Springer Berlin Heidelberg, 2010. doi: 10.1007/3-540-39963-1_23.

[99] D. Holton. A PEPA specification of an industrial production cell. *The Computer Journal*, 38(7):542–551, 1995. doi: 10.1093/comjnl/38.7.542.

[100] J. Hooman. Projectplan ALLEGIO - Composable Embedded Systems for Healthcare. http://www.commit-nl.nl/sites/default/files/Projectplan ALLEGIO Composable Embedded Systems for Healthcare.pdf. Last accessed 2016-12-09.

[101] J. Hu, J. Lygeros, and S. Sastry. Towards a Theory of Stochastic Hybrid Systems. In *Hybrid Systems: Computation and Control*, volume 1790 of *Lecture Notes in Computer Science*, pages 160–173. Springer, 2000. doi: 10.1007/3-540-46430-1_16.

[102] M. N. Huhns and M. P. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 9(1):75–81, 2005. doi: 10.1109/mic.2005.21.

[103] G. Igna, V. Kannan, Y. Yang, A. 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 *Lecture Notes in Computer Science*, pages 170–187. Springer, 2008. doi: 10.1007/978-3-540-85778-5_13.

[104] G. Igna and F. Vaandrager. Verification of printer datapaths using timed automata. In *Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6416 of *Lecture Notes in Computer Science*, pages 412–423. Springer, 2010. doi: 10.1007/978-3-642-16561-0_38.

[105] R. Ihaka and R. Gentleman. R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics*, 5:299–314, 1996. doi: 10.1080/10618600.1996.10474713.

[106] Intel. Intel Ark. `http://ark.intel.com/`. Last accessed 2016-12-09.

[107] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, 1991. ISBN: 978-0-471-50336-1.

[108] K. Jensen, L. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *International Journal on Software Tools for Technology Transfer*, 9(3-4):213–254, 2007. doi: 10.1007/s10009-007-0038-x.

[109] Z. Jiang, M. Pajic, and R. Mangharam. Cyber-physical modeling of implantable cardiac medical devices. *Proceedings of the IEEE*, 100(1):122–137, 2012. doi: 10.1109/jproc.2011.2161241.

[110] P. Jiménez, F. Thomas, and C. Torras. 3D collision detection: a survey. *Computers and graphics*, 25(2):269–285, 2001. doi: 10.1016/s0097-8493(00)00130-8.

[111] H. Jinfeng, J. Voeten, M. Groothuis, J. Broenink, and H. Corporaal. A model-driven design approach for mechatronic systems. In 7^{th} *Int. Conference on Application of Concurrency to System Design*, pages 127–136. IEEE, 2007. doi: 10.1109/acsd.2007.40.

[112] J. Johnson. *Designing with the Mind in Mind: Simple Guide to Understanding User Interface Design Rules*. Elsevier, 2010. ISBN: 978-0-124-07914-4.

[113] B. Jonsson, S. Perathoner, L. Thiele, and W. Yi. Cyclic dependencies in modular performance analysis. In *Proceedings of the* 8^{th} *ACM international conference on Embedded software*, pages 179–188. ACM, 2008. doi: 10.1145/1450058.1450083.

[114] C. Juiz and R. Puigjaner. Queueing Analysis of Pools in Soft Real-Time Systems. In *Computer Performance Evaluation / TOOLS*, volume 1786 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2000. doi: 10.1007/3-540-46429-8_5.

[115] T. Kanungo, D. Mount, N. Netanyahu, C. Piatko, R. Silverman, and A. Wu. An efficient K-means clustering algorithm: analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(7):881–892, 2002. doi: 10.1109/tpami.2002.1017616.

[116] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, 2003. doi: 10.1109/jproc.2002.805824.

[117] S. Keshishzadeh, A. Mooij, and M. Mousavi. Early Fault Detection in DSLs Using SMT Solving and Automated Debugging. In *Software Engineering and Formal Methods*, volume 8137 of *Lecture Notes in Computer Science*, pages 182–196. Springer, 2013. doi: 10.1007/978-3-642-40561-7_13.

[118] S. Keshishzadeh and A. J. Mooij. Formalizing and testing the consistency of DSL transformations. *Formal Aspects of Computing*, 28(2):181–206, 2016. doi: 10.1007/s00165-016-0359-1.

[119] 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 *Lecture Notes in Computer Science*, pages 18–37. Springer, 2002. doi: 10.1007/3-540-45874-3_2.

[120] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf. An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures. In

Proceedings IEEE International Conference on Application-Specific Systems, Architectures and Processors, pages 338–349. IEEE Computer Society, 1997. doi: 10.1109/asap.1997.606839.

[121] K. Kontogiannis, G. Lewis, D. Smith, M. Litoiu, H. Muller, S. Schuster, and E. Stroulia. The landscape of service-oriented systems: A research perspective. In *Proceedings of the International Workshop on Systems Development in SOA Environments*. IEEE Computer Society, 2007. doi: 10.1109/sdsoa.2007.12.

[122] T. J. Koo, B. Sinopoli, A. Sangiovanni-Vincentelli, and S. Sastry. A formal approach to reactive system design: unmanned aerial vehicle flight management system design example. In *Computer Aided Control System Design, 1999. Proceedings of the 1999 IEEE International Symposium on*, pages 522–527. IEEE, 1999. doi: 10.1109/cacsd.1999.808702 .

[123] J. Koomey. Growth in data center electricity use 2005 to 2010. *Oakland, CA: Analytics Press. August*, 1, 2011.

[124] H. Koziolek and R. Reussner. A Model Transformation from the Palladio Component Model to Layered Queueing Networks. In *Performance Evaluation: Metrics, Models and Benchmarks*, volume 5119 of *Lecture Notes in Computer Science*, pages 58–78. Springer, 2008. doi: 10.1007/978-3-540-69814-2_6.

[125] N. Kroes. Using ICT to build Sustainable Cities, Brussels, 2012.

[126] I. Krüger. Specifying Services with UML and UML-RT: Foundations, Challenges and Limitations. *Electronic Notes in Theoretical Computer Science*, 65(7):34–50, 2002. doi: 10.1016/s1571-0661(04)80483-3.

[127] P. Küngas. Petri Net Reachability Checking Is Polynomial with Optimal Abstraction Hierarchies. In *Abstraction, Reformulation and Approximation*, volume 3607 of *Lecture Notes in Computer Science*, pages 149–164. Springer, 2005. doi: 10.1007/11527862_11.

[128] 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. doi: 10.1007/978-3-642-22110-1_47.

[129] M. Kwiatkowska, G. Norman, R. Segala, and J. Sproston. Automatic verification of real-time systems with discrete probability distributions. *Electronic Notes in Theoretical Computer Science*, 282(1):101–150, 2002. doi: 10.1016/s0304-3975(01)00046-9.

[130] M. Kwiatkowska, G. Norman, J. Sproston, and F. Wang. Symbolic Model Checking for Probabilistic Timed Automata. In *Joint International Conferences on Formal Modeling and Analysis of Timed Systmes and Formal Techniques in Real-Time and Fault -Tolerant Systems*, volume 3253 of *Lecture Notes in Computer Science*, pages 293–308. Springer, 2004. doi: 10.1007/978-3-540-30206-3_21.

[131] E. Larsen, S. Gottschalk, M. Lin, and D. Manocha. Fast distance queries with rectangular swept sphere volumes. In *Proceedings of International Conference on Robotics and Automation*, volume 4, pages 3719–3726. IEEE, 2000. doi: 10.1109/robot.2000.845311.

[132] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997. doi: 10.1007/s100090050010.

[133] A. Law and D. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, 1982. ISBN: 978-0-072-98843-7.

[134] I. Lee, J. Y. Leung, and S. H. Son. *Handbook of real-time and embedded systems*. CRC Press, 2007. ISBN: 978-158-488-678-5.

[135] I. Lee, G. Pappas, R. Cleaveland, J. Hatcliff, B. Krogh, P. Lee, H. Rubin, and L. Sha. High-Confidence Medical Device Software and Systems. *IEEE Computer*, 39(4):33–38, 2006. doi: 10.1109/mc.2006.127.

[136] A. Legay and B. Delahaye. Statistical Model Checking : An Overview. *Computing Research Repository*, abs/1005.1327, 2010. doi: 10.1007/978-3-642-16612-9_11.

[137] E. P. Leite. *Matlab Modelling Programming and Simulations*. Sciyo, 2010. ISBN: 978-953-307-125-1.

[138] Y. Liu and H. Zhu. A survey of the research on power management techniques for high-performance systems. *Software: Practice & Experience*, 40(11):943–964, 2010. doi: 10.1002/spe.952.

[139] W. Lo and S. Puchalski. Digital image processing. *Veterinary Radiology & Ultrasound*, 49:S42–S47, 2008. doi: 10.1111/j.1740-8261.2007.00333.x.

[140] L. Lunesu. Find And Replace Text command line utility. `http://fart-it.sourceforge.net/`. Last accessed 2016-12-09.

[141] MathWorks. Matlab. `https://www.mathworks.com/products/matlab.html`. Last accessed 2016-12-09.

[142] M. Mernik, J. Heering, and A. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005. doi: 10.1145/1118890.1118892.

[143] N. Metropolis and S. Ulam. The Monte carlo method. *Journal of the American statistical association*, 44(247):335–341, 1949. doi: 10.1080/01621459.1949.10483310.

[144] Modest. The Modest Toolset. `http://www.modestchecker.net/`. Last accessed 2016-12-09.

[145] A. Mooij, J. Hooman, and R. Albers. Early Fault Detection Using Design Models for Collision Prevention in Medical Equipment. In *Foundations of Health Information Engineering and Systems*, volume 8315 of *Lecture Notes in Computer Science*, pages 170–187. Springer, 2013. doi: 10.1007/978-3-642-53956-5_12.

[146] J. Nolan. "An Inconvenient Truth" Increases Knowledge, Concern, and Willingness to Reduce Greenhouse Gases. *Environment and Behavior*, 42(5):643–658, 2010. doi: 10.1177/0013916509357696.

[147] G. Norman and V. Shmatikov. Analysis of probabilistic contract signing. *Journal of Computer Security*, 14(6):561–589, 2006. doi: 10.3233/jcs-2006-14604.

[148] OMG. Business Process Model and Notation (BPMN). `http://www.omg.org/spec/BPMN/2.0`. Last accessed 2016-12-09.

[149] 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 *Proceedings 18th Real-Time and Embedded Technology and Applications Symposium*, pages 173–184. IEEE, 2012. doi: 10.1109/rtas.2012.25.

[150] M. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-Oriented Computing: State of the Art and Research Challenges. *IEEE Computer*, 40(11):38–45, 2007. doi: 10.1109/mc.2007.400.

[151] PEPA. Performance Evaluation Process Algebra. `http://www.dcs.ed.ac.uk/pepa/`. Last accessed 2016-12-09.

[152] PEPA. Performance Evaluation Process Algebra - Plug-in. `http://www.dcs.ed.ac.uk/pepa/tools/plugin/`. Last accessed 2016-12-09.

[153] S. Perathoner, E. Wandeler, . Thiele, A. Hamann, S. Schliecker, R. Henia, R. Racu, R. Ernst, and M. G. Harbour. Influence of different system abstractions on the performance analysis of distributed real-time systems. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 193–202. ACM, 2007. doi: 10.1145/1289927.1289959.

[154] Peregrine. Peregrine HPC cluster. `https://redmine.hpc.rug.nl/redmine/projects/peregrine`. Last accessed 2016-12-09.

[155] B. Pérez, E. Stafford, J. L. Bosque, and R. Beivide. Energy efficiency of load balancing for data-parallel applications in heterogeneous systems. *The Journal of Supercomputing*, pages 1–13, 2016. doi: 10.1007/s11227-016-1864-y.

[156] A. Philippou, C. Georghiou, and G. Philippou. A generalized geometric distribution and some of its properties. *Statistics & Probability Letters*, 1(4):171–175, 1983. doi: 10.1016/0167-7152(83)90025-1.

[157] Philips Healthcare. Innovating meaningful healthcare | Philips Healthcare. `http://www.usa.philips.com/healthcare`. Last accessed 2016-12-09.

[158] Philips Healthcare. Interventional X-ray systems. `http://www.usa.philips.com/healthcare/solutions/interventional-xray/ixr-systems`. Last accessed 2016-12-09.

[159] Philips Healthcare. Interventional X-ray tools. `http://www.usa.philips.com/healthcare/solutions/interventional-xray/ixr-interventional-tools`. Last accessed 2016-12-09.

[160] A. Pimentel, L. Hertzberger, P. Lieverse, P. van der Wolf, and E. Deprettere. Exploring Embedded-Systems Architectures with Artemis. *IEEE Computer*, 34(11):57–63, 2001. doi: 10.1109/2.963445.

[161] Plotly. Plotly website: Visualize Data, Together. `https://plot.ly/`. Last accessed 2016-12-09.

[162] PMNL. Petri Net Markup Language. `http://www.pnml.org/`. Last accessed 2016-12-09.

[163] R. Pooley. Software engineering and performance: a road-map. In *Proceedings of the Conference on The Future of Software engineering*, pages 189–199. ACM, 2000. doi: 10.1145/336512.336553.

[164] J. Prince and J. Links. *Medical imaging signals and systems*. Pearson Prentice Hall Upper Saddle River, NJ, 2006. ISBN: 978-0-13065-353-6.

[165] PRISM. Probabilistic Symbolic Model Checker. `http://www.prismmodelchecker.org/`. Last accessed 2016-12-09.

[166] P. Puschner and A. Burns. Guest Editorial: A Review of Worst-Case Execution-Time Analysis. *Real-Time Systems*, 18(2-3):115–128, 2000. doi: 10.1023/a:1008119029962.

[167] J. Racine. GNUplot 4.0: a portable interactive plotting utility. *Journal of Applied Econometrics*, 21(1):133–141, 2006. doi: 10.1002/jae.885.

[168] RANDOM.ORG. `https://www.random.org/`, title = Random number generator. Last accessed 2016-12-09.

[169] A. Rantzer. CPN Tools for Editing, Simulating and Analysing Coloured Petri Nets. In *Applications and Theory of Petri Nets 2003: 24th International Conference, ICATPN 2003*, volume 2679, pages 450–462, 2003. doi: 10.1007/3-540-44919-1_28.

[170] W. Reisig. *Understanding Petri Nets*. Springer, 2013. ISBN: 978-3-642-33277-7.

[171] A. Remke and B.R. Haverkort. CSL Model Checking Algorithms for Infinite-State Structured Markov Chains. In *Formal Modeling and Analysis of Timed Systems*, volume 4763 of *Lecture Notes in Computer Science*, pages 336–351. Springer, 2007. doi: 10.1007/978-3-540-75454-1_24.

[172] E. Reschenhofer. Generalization of the Kolmogorov-Smirnov test. *Computational Statistics & Data Analysis*, 24(4):433–441, 1997. doi: 10.1016/s0167-9473(96)00077-1.

[173] D. Ritter. Using the Business Process Model and Notation for Modeling Enterprise Integration Patterns. *Computing Research Repository*, abs/1403.4053, 2014. doi: 10.1007/978-3-319-09195-2_17.

[174] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003. ISBN: 978-0-133-60124-4.

[175] 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 *Lecture Notes in Computer Science*, pages 166–181. Springer, 2012. doi: 10.1007/978-3-642-28540-0_12.

[176] W. Sandmann. Rare Event Simulation Methodologies and Applications. *Simulation*, 83(12):809–810, 2007. doi: 10.1177/0037549708090022.

[177] A. Sangiovanni-Vincentelli and M. Di Natale. Embedded system design for automotive applications. *IEEE Computer*, 40(10):42–51, 2007. doi: 10.1109/mc.2007.344 .

[178] B. Schatz, A. Pretschner, F. Huber, and J. Philipps. Model-Based Development of Embedded Systems. In *Advances in Object-Oriented Information Systems*, volume 2426 of *Lecture Notes in Computer Science*, pages 298–311. Springer, 2002. doi: 10.1007/3-540-46105-1_34.

[179] J. Seward, B. Khandheria, J. Oh, M. Abel, R. Hughes, W. Edwards, B. Nichols, W. Freeman, and A. Tajik. Transesophageal echocardiography: Technique, anatomic correlations, implementation, and clinical applications. *Mayo Clinic Proceedings*, 63(7):649 – 680, 1988.

[180] P. Shahabuddin. Rare Event Simulation in Stochastic Models. In *Winter Simulation Conference*, pages 178–185. IEEE Computer Society, 1995. doi: 10.1109/wsc.1995.478721.

[181] V. Sharma and K. Trivedi. Architecture based analysis of performance, reliability and security of software systems. In *Proceedings of the 5th International Workshop on Software and Performance.*, pages 217–227. ACM, 2005. doi: 10.1145/1071021.1071046.

[182] S. Silberstein. Using Anytime Algorithms in Intelligent Systems. *AI Magazine*, 17(3):73–83, 1996.

[183] M. Sonka, V. Hlavac, and R. Boyle. *Image processing, analysis, and machine vision*. Cengage Learning, 2014. ISBN: 978-049-508-252-1.

[184] G. Steel. Formal analysis of PIN block attacks. *Electronic Notes in Theoretical Computer Science*, 367(1-2):257–270, 2006. doi: 10.1016/j.tcs.2006.08.042.

[185] S. Stuijk, M. Geilen, and T. Basten. SDF3: SDF For Free. In 6th *International Conference on Application of Concurrency to System Design*, pages 276–278. IEEE Computer Society, 2006. doi: 10.1109/acsd.2006.23.

[186] STW. STW project Cooperative Networked Systems. `https://www.utwente.nl/ctit/research/research_projects/national/stw/cns/`. Last accessed 2016-12-09.

[187] R. Tawhid and D. Petriu. User-friendly approach for handling performance parameters during predictive software performance engineering. In *Proceedings of the 3rd International Conference on Performance Engineering*, pages 109–120. ACM, 2012. doi: 10.1145/2188286.2188304.

[188] B. Theelen, M. Geilen, A. Basten, J. Voeten, S. Gheorghita, and S. Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *4th ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, pages 185–194. IEEE Computer Society, 2006. doi: 10.1109/memcod.2006.1695924.

[189] B. Theelen, J. Voeten, and R. Kramer. Performance modelling of a network processor using POOSL. *Computer Networks*, 41(5):667–684, 2003. doi: 10.1016/s1389-1286(02)00455-3.

[190] The Möbius team. The Möbius tool. `https://www.mobius.illinois.edu/`. Last accessed 2016-12-09.

[191] The Palladio team. Palladio Component Model. `http://www.palladio-simulator.com/science/palladio_component_model/`. Last accessed 2016-12-09.

[192] TNO-ESI. Model-Driven Design-Space Exploration: The Octopus Toolset. `http://dse.esi.nl/`. Last accessed 2016-12-09.

[193] J. M. Torpy. Percutaneous coronary intervention. *JAMA*, 291(6):778, 2004. doi: 10.1001/jama.291.6.778.

[194] N. Trcka, M. Hendriks, A. Basten, M. Geilen, and L. Somers. Integrated model-driven design-space exploration for embedded systems. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 339–346. IEEE, 2011. doi: 10.1109/samos.2011.6045482.

[195] J. Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer networks and ISDN systems*, 29(1):49–79, 1996. doi: 10.1016/s0169-7552(96)00017-7.

[196] M. Tribastone. A Fluid Model for Layered Queueing Networks. *IEEE Trans. Software Eng.*, 39(6):744–756, 2013. doi: 10.1109/tse.2012.66.

[197] W. Tsai. Service-Oriented System Engineering: a new paradigm. In *Service-Oriented System Engineering, IEEE International Workshop*, pages 3–6. IEEE, 2005. doi: 10.1109/sose.2005.34.

[198] A. Tucker. *Applied Combinatorics*. Wiley, 1980. ISBN: 978-111-821-011-6.

[199] University of Dortmund. The Hierarchical Evaluation Tool HIT. `http://ls4-www.cs.tu-dortmund.de/HIT/HIT.html`. Last accessed 2016-12-09.

[200] Uppaal. Uppsala Aalborg model checker. `http://www.uppaal.org/`. Last accessed 2016-12-09.

[201] F. van den Berg. The official iDSL homepage. `http://www-i-dsl.org`. Last accessed 2016-12-09.

[202] F. van den Berg, B.R. Haverkort, and J. Hooman. Efficiently Computing Latency Distributions by Combined Performance Evaluation Techniques. In *Proceedings of the 9th EAI International Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS'15, pages 158–163. ICST, 2015. doi: 10.4108/eai.14-12-2015.2262725.

[203] 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. doi: 10.1007/978-3-319-23267-6_14.

[204] F. van den Berg, B. Postema, and B.R. Haverkort. Evaluating load balancing policies for performance and energy-efficiency. In *Quantitative Aspects of Programming Languages and Systems*, volume 227 of *Electronic Proceedings in Theoretical Computer Science*, pages 98–117, 2016. doi: 10.4204/eptcs.227.7.

[205] 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. doi: 10.4230/OASIcs.MCPS.2014.80.

[206] 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. doi: 10.1007/978-3-319-23267-6_15.

[207] 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 *Lecture Notes in Computer Science*, pages 276–287. Springer, 2013. doi: 10.1007/978-3-642-40725-3_21.

[208] A. Van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *Sigplan Notices*, 35(6):26–36, 2000. doi: 10.1145/352029.352035.

[209] J. P. Voeten. Performance evaluation with temporal rewards. *Performance Evaluation*, 50(2):189–218, 2002. doi: 10.1016/s0166-5316(02)00105-0.

[210] T. Waizmann and M. Tribastone. DiffLQN: Differential Equation Analysis of Layered Queuing Networks. In *International Conference on Performance Engineering*, pages 63–68. ACM, 2016. doi: 10.1145/2859889.2859896.

[211] M. Walter, A. Gouberman, M. Riedl, J. Schuster, and M. Siegle. LARES: A novel approach for describing system reconfigurability in dependability models of fault-Tolerant systems. In *Proceedings of European Safety and Reliability Conference*, pages 153–160, 2009. doi: 10.1201/9780203859759.ch22.

[212] E. Wandeler, L. Thiele, M. Verhoef, and P. L. System architecture evaluation using modular performance analysis: a case study. *International Journal on Software Tools for Technology Transfer* , 8(6):649–667, 2006. doi: 10.1007/s10009-006-0019-5.

[213] S. Wang and K. Shin. Early-stage performance modeling and its application for integrated embedded control software design. *ACM Software Engineering Notes*, 29(1):110–114, 2004. doi: 10.1145/974043.974061.

[214] J. Wilson. Gantt charts: A centenary appreciation. *European Journal of Operational Research*, 149(2):430–437, 2003. doi: 10.1016/s0377-2217(02)00769-5.

[215] M. Woodside, G. Franks, and D. Petriu. The Future of Software Performance Engineering. In *2007 Future of Software Engineering*, pages 171–187, Washington, DC, USA, 2007. IEEE Computer Society. doi: 10.1109/fose.2007.32.

[216] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang. A smart hill-climbing algorithm for application server configuration. In *Proceedings of the 13th International Conference on World Wide Web*, pages 287–296. ACM, 2004. doi: 10.1145/988672.988711.

[217] Y. Yang. Ensemble learning. In *Temporal Data Mining Via Unsupervised Ensemble Learning*, pages 35–56. Elsevier, 2017. doi: 10.1016/b978-0-12-811654-8.00004-x.

[218] R. Zurawski. *Embedded Systems Handbook*. CRC Press, 2005. ISBN: 978-142-007-410-9.