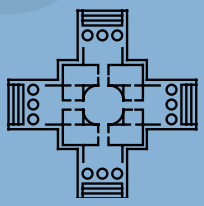
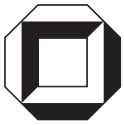
The Karlsruhe Series on
Software Design
and Quality

2



Parameter Dependencies for Reusable Performance Specifications of Software Components

Heiko Koziolek



universitätsverlag karlsruhe

Heiko Koziolek

Parameter Dependencies for Reusable Performance Specifications of Software Components

The Karlsruhe Series on Software Design and Quality

Volume 2

Chair Software Design and Quality
Faculty of Computer Science
Universität Karlsruhe (TH)

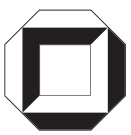
and

Software Engineering Division
Research Center for Information Technology (FZI), Karlsruhe

Editor: Prof. Dr. Ralf Reussner

Parameter Dependencies for Reusable Performance Specifications of Software Components

by
Heiko Koziolek



universitätsverlag karlsruhe

Dissertation, University of Oldenburg,
Department of Computer Science, 2008

Impressum

Universitätsverlag Karlsruhe
c/o Universitätsbibliothek
Straße am Forum 2
D-76131 Karlsruhe
www.uvka.de

Universitätsverlag Karlsruhe 2008
Print on Demand

ISSN: 1867-0067
ISBN: 978-3-86644-272-6

Contents

List of Figures

List of Tables

Abstract

Despite the increasing computational power of modern computers, many large, distributed software systems still suffer from performance problems today. To avoid design-related performance problems, model-driven performance prediction methods analyse the response times, throughputs, and resource utilisations of systems under development based on design documents before and during implementation. For component-based software systems, existing prediction methods neglect the performance influence of different usage profiles (i.e., the number of requests and the included parameter values) in their specification languages, which limits their prediction accuracy. This thesis proposes new modelling languages and according model transformations, which allow a reusable description of usage profile dependencies in component-based software systems. The thesis includes an experimental evaluation, which shows that predictions based on the newly introduced models can support design decisions for scenarios, whose performance is influenced by different usage profiles.

Zusammenfassung

Trotz der ständig ansteigenden Leistung moderner Rechner leiden auch heute noch viele verteilte, betriebliche Anwendungssysteme unter Performance-Problemen. Eine häufige Ursache dafür sind Defizite im Entwurf der Software solcher System. Um diese Defizite zu vermeiden, analysieren modellgetriebene Performance-Vorhersageverfahren die Antwortzeiten, Durchsätze und Ressourcenauslastungen von neu zu entwickelnden Systemen schon bevor bzw. während ihrer Implementierung auf Basis von Entwurfsdokumenten. Existierende Vorhersageverfahren für komponentenbasierte Softwaresystemen vernachlässigen dabei performancerelevante Einflüsse durch unterschiedliche Benutzungsprofile (bestehend aus der Anzahl von Anfragen und deren enthaltenen Parameterwerten). Durch diese Ungenauigkeit bei der Modellierung sinkt die Vorhersagegenauigkeit dieser Verfahren. Daher schlägt diese Dissertation neue Modellierungssprachen und darauf aufbauende Modelltransformationen vor, die eine wiederverwendbare Beschreibung von Benutzungsprofilabhängigkeiten in komponentenbasierten Softwaresystemen erlauben. Mit einer experimentellen Untersuchung zeigt diese Arbeit, dass Vorhersagen basierend auf den neuen Modellen Entwurfsentscheidungen von Software-Architekten insbesondere in solchen Fällen unterstützen können, in denen Benutzungsprofile die Performance geplanter Systeme beeinflussen.

Acknowledgements

Although a PhD thesis has only one author, there are actually many people involved in its creation. I'd like to thank the people, who were influential to my research during the last three years and enabled me to write this thesis.

First of all, I thank my parents for their absolute support over the years, which greatly contributed to my work. Second, my supervisors Ralf Reussner and Wilhelm Hasselbring enabled me to conduct my research and provided professional support. Ralf laid the foundation for my research, put lots of confidence in me, and created a very enjoyable and fruitful working atmosphere. Willi managed the graduate school I participated in and provided feedback during our PhD seminars and reading groups.

The members of the DFG-group Palladio and the Chair Software Design & Quality (SDQ) from the University of Karlsruhe helped me through many discussions, which shaped my topic and greatly increased the quality of my research. Especially Steffen Becker and Jens Happe had a major influence on my work. Our challenging discussions, mutual detailed feedback, constructive criticism, and respectful teamwork were a tremendous help for me. Viktoria Firus, Klaus Krogmann, and Michael Kuperberg from Palladio and SDQ, as well as Thomas Goldschmidt, Henning Groenda, Christoph Rathfelder, and Johannes Stammel from Forschungszentrum Informatik helped me with their comments during PhD seminars, their paper reviews, and their ideas during our research meetings.

I supervised two diploma students, who each provided their piece of the puzzle to my thesis. Thomas Kappler improved my concepts for code analysis and created a prototypical tool. Anne Martens managed the experimental evaluation conducted for this thesis and put lots of effort into making it a success. I thank both of them for their dedication.

During the last three years, I was part of the Graduate School "Trustsoft" at the University of Oldenburg. This created many opportunities for discussions and let me meet many interesting researchers. I thank all members of Trustsoft for their support, especially Marko Boskovic, Simon Giesecke, Henrik Lipskoch, Roland Meyer, Astrid Rakow, Matthias Rohr, Christian Storm, Timo Warns, Ira Wempe, Daniel Winteler, and Manuela Wüstefeld.

Furthermore, I was lucky to discuss my topic with many great researchers. At the risk of leaving someone out, I especially thank Antonia Bertolino, Egor Bondarev, Ivica Crnkovic, Alexandre Fioukov, Vincenzo Grassi, Ian Gorton, Moreno Mar-

zolla, Marcus Meyerhöfer, Raffaela Mirandola, Sven Overhage, Iman Poernomo, Antonino Sabetta, Heinz Schmidt, Connie Smith, Clemens Szyperski, and Murray Woodside.

Finally, I'd like to thank Jorge Cham, the author of PhD-comics, for letting PhD students all over the world bear grad school life with a smile and being a great source for procrastination.

Chapter 1

Introduction

The following introduction will motivate the need for a new modelling method for component-based performance engineering (Chapter 1.1) and then describe the specific problem tackled in this thesis in detail (Chapter 1.2). Afterwards, it will point out the deficits of existing solution approaches to this problem (Chapter 1.3), before it lists the scientific contributions of this thesis (Chapter 1.4). Finally, the introduction will sketch the experimental validation conducted for this thesis (Chapter 1.5).

1.1 Motivation

For more than 30 years, the performance of computer systems has doubled every 24 month (Moore's Law, [Moo65]). Many engineers believe that this trend will continue in the near future [Kan03]. Although processors become faster at an exponential pace, performance problems are still prevalent in many large software systems [WFP07]. Recently, a survey of IT executives reported that 50% of the investigated companies had encountered performance problems with at least 20% of their applications [Com06].

Insufficient performance in large software system is often caused by design flaws [Smi02]. For example, in April 2007 it was reported [hn07] that the development of SAP's software solution for medium-sized businesses (A1S) caused problems because of insufficient performance. The estimated overall development costs for the project were between 300 and 400 million Euros. An early implementation of the system was only able to handle up to 10 concurrent users instead of the targeted 1000 concurrent users because of design flaws that prevented the system from using different servers.

Many software developers have a "fix-it-later" attitude [Smi02] towards the performance of their systems. They first focus on implementing the system's functionality without regarding performance requirements. Later, they test and optimise the performance after they have completed the implementation [SMF+07]. However, if performance problems of a system result from design flaws, re-design and subsequent re-implementation can become very expensive. Williams et al. [WS03] estimate the costs for re-design due to performance problems in late development stages to several million US-dollars even in typical mid-sized software projects.

Additionally, it is often not possible to overcome performance problems with additional hardware ("kill-it-with-iron"), because a software architecture can exhibit bottlenecks preventing the performance to scale up linearly with the available hardware [Smi02]. Such systems do not benefit from the increasing computational power of new hardware systems projected by Moore's Law.

Model-driven performance prediction [BDIS04] proposes a substantially less expensive solution to this problem [WS03]. This method (cf. Fig. 1.1) uses software models (e.g., in UML [Obj07b]) and lets developers annotate them with performance properties based on estimations or measurements (e.g., with the UML SPT profile [Obj05b]). Transformation tools then map these annotated software models to performance models, such as queueing networks or Petri nets. These approaches are called "model-driven" instead of "model-based", because they automate the transformation process.

For the resulting performance models different solvers based on mathematical analysis or simulation are available. They derive performance metrics, such as the response time, throughput, or resource utilisation, given a certain workload. Due to the involved estimations and the uncertainty during early development stages, these are not guaranteed real-time predictions. Nevertheless, software architects can use the prediction results to assess the maximum performance of a system during design and identify bottlenecks without an implementation.

Since the advent of component-based software engineering (CBSE) [SGM02], researchers propose performance prediction methods specifically for component-based systems [BGMO06]. A software component is a contractually specified building block for a software system with explicit provided and required interfaces [SGM02]. Component developers shall specify the performance of their components individually, which limits the complexity and therefore the effort for modelling. Software architects shall compose these specifications isomorphically to the desired architecture and then derive performance metrics from the resulting model

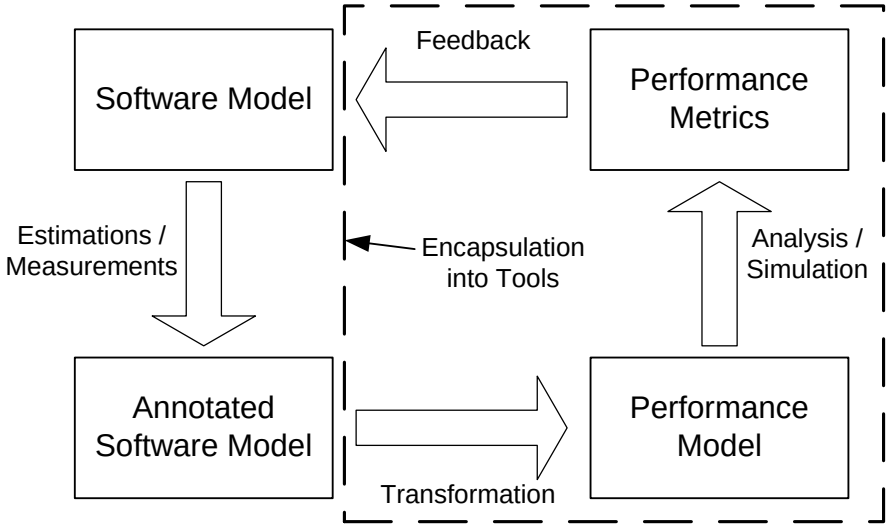


Figure 1.1: Model-Driven Performance Prediction

supported by tools [BM04a, WW04]. As in other engineering disciplines, these methods shall enable reasoning on the properties of a system based on the properties of its individual parts [RPS03].

However, specifying the performance properties of a software component is not trivial [BGMO06]. Component developers cannot make assumptions on how the component will be composed with other components [FBH05], thus they do not know how required services will influence the performance properties of their components. They do not know the hardware resources and properties of the middleware in the customer's environment [WVCB01]. They cannot foresee contention delays, if a component is used concurrently or runs in parallel with other applications on the same hardware [BGMO06]. Finally, component developers do not know how the component will be *used* by clients, which can alter the execution time [HMW04].

While recognising each of these influence factors on the performance of a software component, this thesis especially focusses on the latter factor, i.e., the problem of reflecting usage dependencies in component performance specifications.

1.2 Problem

Component developers shall specify the performance of their components, but do not know how clients will use them. They implement a component against interface specifications [SGM02], which provide no information how often and with

what parameter values clients will use it. The usage profile (i.e., the number of requests and the included parameter values) can alter performance properties significantly [HMW04]. For example, the execution time of a component processing items in a list heavily depends on the size of the list. Some clients might use only small lists, while others use large lists. In some systems, a low number of clients might use the component infrequently, while in other systems, many clients use it constantly. Software components are meant for reuse in different contexts, therefore highly variable usage is possible [SGM02].

Input parameters do not only alter a component's use of hardware resources, they can also change the interaction with required services [HMW04]. Calling a component service with different parameter values can lead to different required services being called. In other cases, the number of calls to the same required service can depend on input parameters. The component developer's performance specification has to reflect this, as calls to required services alter the response times perceived at a component's provided interfaces.

The parameter values of clients calling a component can also alter the parameter values the component uses to call required services. This is for example common if a component is used in a pipe-and-filter pattern, where it receives some input, transforms it, and then submits it to another component. Furthermore, the return values from calls to required services add to the usage profile of a component and might again alter resource usage or usage of other required services. As the input parameters by clients, component developers cannot make assumptions on the return values of calls to required services.

Parameter values can also alter the internal state of a component [HMW04]. Depending on its internal state, a component can again use resources or required services differently. Therefore, a component performance specification language should provide facilities to express the internal state of a component in dependency to parameter values.

Developing a modelling language for component performance specification that includes usage profile dependencies has to consider the tradeoff between model expressiveness and tractability [Jai91]. Complex models are often not suited for performance predictions, because they suffer from state space explosion, which leads to mathematical intractability. Therefore, performance models should be as abstract as possible, while on the other hand still be sufficiently detailed to allow accurate predictions.

1.3 Existing Solutions

To solve the problems described before, component developers need a modelling language that allows the specification of performance properties in dependency to parameter values. The specification must be a function allowing the characterisation of parameter dependencies both to resources usages and calls to required services. Existing approaches for component performance specification and prediction are either (i) related to complexity theory, or (ii) based on UML, or (iii) use their own notations.

A performance specification language for software components based on *complexity theory* has been proposed by Sitaraman et al. [SKK+01]. According to this approach, component developers shall use extended Big-O notations to specify the algorithmic complexity of component services. While including parameter dependencies to resource usages, this approach models performance on a very high abstraction level. It is not possible to get quantitative performance metrics, such as execution time, from the resulting specifications, but only the complexity class. This information is usually not sufficient for software architects, who want to evaluate performance requirements. The approach also neglects calls to required services and their parameterisation.

Many model-driven performance prediction approaches for monolithic system are based on the *UML* and the UML SPT profile [Obj05b] standardised by the Object Management Group (OMG). While the UML supports modelling software components, there is no special support for component performance specifications. The SPT profile does not include a notion of input/output parameter values, therefore it is generally not possible to express parameter dependencies. The CB-SPE method [BM04a] is a performance prediction approach for component-based systems based on the UML SPT profile and does not deal with parameter dependencies.

Other approaches for component-based performance prediction use *proprietary notations* (e.g., [HMW04, WW04, BdWCM05, EFH04]). Hamlet et al. [HMW04] tackle the problem of parameter propagations, but use a very restricted component model and for example do not parameterise resource demands. Wu et al. [WW04] use the Component-based Modelling Language (CBML), which is based on layered queueing networks. It does not include a notion of parameters from component interfaces. Bondarev et al. [BdWCM05] propose a specification language based on the ROBOCOP component model [GL03]. It allows parameterisation of resource usages

and calls to required services, but models only constant parameter values instead of probability distributions, which restricts expressiveness. Eskenazi et al. [EFH04] parameterise models in an ad-hoc manner, which limits their reusability. None of these proprietary approaches has produced industry-relevant tools.

Several other prediction approaches have modelled the performance properties of component-based software systems (e.g. [GM01, HMSW02, Kou06, CLGL05, LFG05]. However, these approaches have different notions of software components, use monolithic models, or do not target reusable specifications. Some of these approaches also heavily rely on performance measurements and require a prototype implementation of the architecture.

The usage of software systems by clients is modelled differently in different areas of computer science. In performance engineering [Jai91], often only the number of users concurrently present in the system is modelled explicitly. The influence of parameter values is usually not expressed explicitly, as many approaches include execution times for a fixed set of parameter values in their models. In software testing, developers sometimes use an operational profile [Mus93], which simply assigns call probabilities to the functions of a system. In reliability engineering, some approaches use Markov usage models [WP93], which model call sequences and their probabilities. None of these approaches explicitly deals with parameter values.

Concluding, existing performance modelling languages only provide limited support for modelling usage profile dependencies. Therefore, their expressiveness and their prediction accuracy is limited in situations where the usage profile influences performance properties.

1.4 Contributions

This thesis proposes the following contributions to the body of knowledge in software engineering:

- **RDSEFF Modelling Language:** The resource demanding service effect specification (RDSEFF) is a new modelling language to abstractly describe the performance properties of software component services. It explicitly allows the specification of resource demands and required service calls in dependency to abstract parameter characterisations. Therefore, it is more expressive than former modelling languages for component performance and enables more accurate predictions. It has been implemented as a meta-model and become part of the

Palladio Component Model (PCM) [RBH+07], which is based on the concept of parameterised contracts introduced by Reussner et al. [RPS03]. A mapping to queueing Petri nets (QPN) specifies the performance-related semantics of the RDSEFF language. This model has been published in [BKR07, KBH07, BKR08].

- **Usage Modelling Language:** The PCM Usage Model is a modelling language, which describes the user interaction with a component-based system. It specifically targets performance predictions and allows the specification of the workloads and parameter characterisations supplied by users. It is more expressive than Markov usage models. The Usage Model is implemented as a meta-model and part of the PCM. It has been published in [BKR07, BKR08].

- **Parameter Characterisation Model:** This thesis introduces a new method for characterising parameter values. The parameter characterisation model added to the PCM allows modelling parameter values and parameter meta-data with random variables. Therefore, domain experts can model the interaction of large user groups with a software system in a stochastic manner. It is more expressive than using constant parameter values in the performance models. To solve the parameter dependencies in RDSEFFs the so-called Stochastic Expression Framework [RBH+07] allows calculating arithmetic operations on random variables. The characterisation model has first been published in [KHB06] as an extension to the UML SPT profile, and later became part of the PCM [RBH+07, BKR08].

- **QoS-Driven Component-based Development Process Model:** The PCM targets a specific development process model derived from the component-based development model by Cheesman et al. [CD01]. As part of this thesis, this model has been extended for performance prediction. It introduces the concept of dividing the performance meta-model among the different roles developing a component-based software system. This allows restricted, domain-specific modelling languages for specific developer roles, which only capture the information available to an individual role. Therefore, the roles can models their parts of a system independently. Other than similar process models, it explicitly includes the roles of the component deployer and the domain expert (published in [KH06]).

- **Extensions to the SRE-Model:** As a performance model capable of deriving response time prediction from PCM instance, the Palladio research group has

developed the Stochastic Regular Expression (SRE) model [FBH05]. It is able to handle general distributed execution times from PCM instances, but only supports single user scenarios. In this thesis, the model has been extended with a new loop concept (published in [KF06]), which allows more accurate predictions. Furthermore, a model transformation from PCM instances to SRE instances has been implemented as part of this thesis (published in [KBH07]).

- **Model Transformation to LQNs:** To connect an analytical solver for multi-user scenarios to the PCM, this thesis provides a mapping of PCM instances to layered queueing networks (LQN) [FMN+96]. LQNs are a popular performance model for distributed systems. They do not support predictions involving general distribution functions, but their analytical solver is substantially more efficient than a simulation of PCM instances (cf. [Bec08]) but it only allows mean-value analysis. Besides the additional solver, the mapping developed in this thesis also allows comparing the expressiveness of the PCM language with the LQN language. The mapping has been implemented prototypically and validated with the model of a distributed system. This work has not been published yet.

- **RDSEFF Generation with Java2PCM:** For implemented software components, an automation of the performance modelling process is desirable. It is possible to partially derive RDSEFFs from source code using static code analysis. This enables fast model creation and lowers the barrier to use the proprietary PCM modelling language. In the context of this thesis, a prototypical analysis of Java code has been implemented, which creates RDSEFF models from Java source files (Java2PCM) [Kap07]. It automatically performs the RDSEFF's abstractions on arbitrary Java code given the component boundary specifications. It does not derive resource demands, which required dynamic analysis. A validation on a larger component-based system (CoCoME [RRMP08]) shows, that the tool is capable of reducing the manual effort for modelling substantially, while providing the same prediction accuracy with the generated models (published in [KKKR08]).

Notice that the concept of usage profile propagation using parameter dependencies developed in this thesis is not restricted to performance modelling. It is possible to analyse other compositional QoS-attributes (e.g., reliability, availability) with only small adaptions of the modelling language (also see [HMW04]). Different QoS-attributes could be combined in a single specification language to analyse their

dependencies (e.g., performability analysis). This has not been attempted in this thesis and is regarded as future work.

This work is related to the PhD theses of Steffen Becker [Bec08], Jens Happe [Hap08], and Klaus Krogmann. Appendix A shows the relationships between these works.

1.5 Validation

To evaluate the claimed benefits of the proposed modelling languages and model transformations, they have been implemented and applied on distributed, component-based systems.

The implementation of the PCM meta-models uses the Ecore modelling language from the Eclipse Modelling Framework (EMF) [Eclb] as a meta-meta-model. OCL constraints in the PCM meta-model specify rules for well-formedness of PCM model instances. Becker [Bec08] has implemented several graphical model editors with the Graphical Modelling Framework (GMF) [Eclc] to create the individual parts of a PCM instance (e.g., RDSEFF, Usage Model, etc.).

Besides the PCM, also the performance models SRE and LQN have been implemented as Ecore meta-models as part of this thesis. The implementation of the model transformations from PCM instances to these models uses the Java programming language. Also a tool for removing parameter dependencies from RDSEFF instances (called Dependency Solver) and the SRE solver have been implemented in Java. The solution of LQN instances relies on existing LQN solvers by the RADS research group from Carleton University, Canada [Rea].

The implementation of the languages enabled the modelling and analysis of component-based systems to assess the achievable prediction accuracy. It is not possible to formally prove that analysis results with the models correctly predict the performance of the implemented system. During early development stages, the information encoded in the model instances is still subject to uncertainty. Therefore, predictions based on the models only demonstrate the principle feasibility of reaching certain performance goals under the assumption that the implementation does not change the design of the system significantly. To assess prediction accuracy, this thesis contains prediction results based on models and compares them with measurement results based on implementations.

For this thesis, the performance of the so-called "Media Store" system has been predicted in a *case study* (published in [KBH07]). It is an artificial, but representative

component-based system, which resembles the functionality of a simple version of Apple's iTunes Music Store [App]. As a large, distributed system, it is in the range of systems targeted by the PCM. The Media Store allowed analysing the benefits of the newly introduced parameterisation concept to performance specifications. Later in this thesis, the architecture is analysed with two different usage profiles and the prediction results are compared to measurement results of an implementation of the store based on Java EE. The deviation between predictions and measurements was below 10 percent.

To assess the sensitivity of the Media Store to changes in the modelled parameters, this thesis also includes a *sensitivity analysis* of the architecture. Different parameter values (e.g., file sizes, number of users, etc.) were varied individually, and the performance was predicted in each case. The analysis showed that the architecture is most sensitive to the number of users, as a higher number of users quickly increases the response times of the system. Other parameters, such as the file sizes or the speed of the hardware resources are less influential.

In order to assess the practicability of the method and tools developed in this thesis, additionally a *controlled experiment* was conducted [Mar07]. It complemented the validation of the introduced modelling languages. After extensive training, 19 computer science students modelled different component-based systems using the graphical PCM model editors. They predicted the performance of these systems for different usage profiles. The study showed that most students were able to achieve a decent prediction accuracy, but that the tools for modelling parameter dependencies still need improvement.

A validation of the modelling features and transformations introduced in this thesis on a system from industry is still missing and regarded as future work.

1.6 Overview

This thesis is structured as follows:

- **Chapter 2** introduces basics in the areas of component-based software engineering (Chapter 2.1), model-driven software development (Chapter 2.2), and performance engineering (Chapter 2.3) needed for the comprehension of the later concepts. It defines the most important terms from these areas as understood in this thesis. Chapter 2.4 connects these different areas, and explains the specifics of component-based performance engineering. In particular, it lists

several requirements for a component performance specification language. Afterwards, Chapter 2.5 surveys related work to this thesis and classifies it according to the requirements established in the former subsection. It looks in detail at the performance specification languages of related approaches and discusses their concepts for reflecting usage profile dependencies.

- **Chapter 3** provides specific background for the context of the developed modelling languages. In Chapter 3.2, it gives an overview of the Palladio Component Model (PCM), i.e., the parts not contributed in this thesis involving component and interface specification as well as resource environment modelling. Chapter 3.3 explains some basics about random variables and probability distributions, because the behavioural description languages introduced in the following section heavily rely on them. It also includes the Stochastic-Expression framework of the PCM, which allows the specification of probability distributions in PCM instances and provides boolean and arithmetic operations to describe parameter dependencies.

- **Chapter 4** describes the behavioural modelling languages contributed to the PCM by this thesis. Chapter 4.1 details on the parameter characterisation model, Chapter 4.2 on the PCM Usage Model, and Chapter 4.3 on the RDSEFF. Each of these Chapters contains a description of the respective meta-model, informal semantics of each meta-class, and example meta-model instances. Finally, Chapter 4.4 maps PCM instances to hierarchical QPNs to formally describe the performance-related semantics of the newly introduced behavioural modelling languages and provides two examples.

- **Chapter 5** first motivates the need for automatic generation of performance models (Chapter 5.1) and then surveys several methods (Chapter 5.2) including static code analysis, dynamic analysis, prototyping, program slicing, and symbolic execution. Then, in Chapter 5.3, it introduces a hybrid approach consisting of static code analysis and dynamic analysis for the generation of RDSEFFs from code. The static code analysis part of this approach has been implemented during this thesis. Chapter 5.4 explains its main tasks. It maps arbitrary Java code to RDSEFF instances and performs different abstractions on the source. The section concludes with a small case study, where the Java2PCM tools implementing the static code analysis has been applied on a component-based system.

- **Chapter 6** is concerned with model transformations from the software domain to the performance domain. After illustrating the transformation process in Chapter 6.1, it describes a preliminary step of the transformation in this thesis, i.e., the solving of parameter dependencies in PCM instances, in Chapter 6.2. Chapter 6.3 explains the SRE model and its solution and defines a mapping from PCM instances to SRE instances. Finally, Chapter 6.4 describes layered queueing networks (LQN) in depth and shows a mapping from PCM instances to LQN instances.

- **Chapter 7** includes the experimental evaluations described previously. Chapter 7.3 describes the Media Store architecture and the case study based on it. It additionally includes a sensitivity analysis of this system. The controlled experiment investigating the practicability of the PCM is the topic of Chapter 7.4. It explains the design of the study, its conduction, and its results.

- **Chapter 8** concludes the thesis. It provides a summary of the thesis' scientific contributions and their benefits to model-driven performance prediction. The section also lists open questions not tackled in this thesis, which are subject to future research.

Chapter 2

Software Components and Performance: Basics and State-of-the-Art

The models introduced in this thesis allow the specification of performance properties of component-based software systems and aim at increasing the performance prediction accuracy for such systems. This Chapter lays the foundation of the thesis. It explains basics from the different areas touched by this thesis, which are necessary to comprehend and evaluate the later introduced models and transformations.

Chapter 2.1 explains important concepts of component-based software engineering (CBSE). It clarifies the notion of software component, software architecture, and the component-based development process. Chapter 2.2 introduces basic terms of model-driven software development (MDSD), such as domain, meta-model, model, and model transformation. Chapter 2.3 briefly surveys the area of performance engineering (PE). It describes performance metrics, performance analysis methods, and performance models.

After an isolated description of the different areas in Chapter 2.1-2.3, Chapter 2.4 combines their concepts and explains the specifics of component-based performance engineering based on modelling. In particular, it lists the different performance-related influence factors on software components and derives a number of requirements for a specification language for component performance.

Many other approaches have tried to tackle the problem of component-based performance engineering. Chapter 2.5 classifies related work and specifically analyses the usage profile dependencies modelling capabilities of the proposed methods.

2.1 Component-based Software Engineering

This section briefly surveys CBSE and defines the most important terms from this area used in this thesis. Chapter 2.1.1 motivates CBSE and gives some background on the development of the discipline. Chapter 2.1.2 defines the notion of software component, before Chapter 2.1.3 discusses composing components to software architectures. Chapter 2.1.4 finally describes the component-based development process.

2.1.1 Introduction

Motivation The use of independently manufactured components to build complex systems is common in many engineering disciplines [HC01]. For example, introducing components to car manufacturing by Henry Ford as a prerequisite for an assembly line in the early 20th century was a major factor to help cars become a mainstream good. Components allow a division of work between component manufactures and component assemblers, thereby reducing the complexity of their individual tasks and shortening overall production time.

In software engineering, a major goal for the use of software components besides the division of work is reusability [SGM02, p.12]. Other than hardware components, software components can easily be copied, therefore it is desirable to implement a certain functionality once and then reuse it in many different contexts by copying the implementation. Besides reusability, decomposing a system into replaceable components prepares the system for change, as individual parts can be exchanged more easily with newer versions [CD01, p.2]. This is especially beneficial in the IT industry, where systems are evolving and technologies are changing at a rapid pace.

Because of the targeted reuse, software components are usually more thoroughly tested than other software, therefore potentially increasing the quality of a component system [Wey98]. Furthermore, assembling a system from prefabricated components allows compositional reasoning on the properties of the system based purely on the specifications of individual components [RPS03]. This can for example be used to determine extra-functional properties, such as performance, reliability, or security of a component-based system, without writing any code.

History The idea to decompose software systems into replaceable parts is as old as software engineering itself. McIlroy et al. [MBNR68] proposed the term 'software component' at the 1968 NATO conference on software engineering. As a result, he

later included pipe and filters into the Unix operating system as a first implementation of the concept. In 1986, Brad Cox [Cox86] proposed 'software ICs' as a modern concept of software components and started to create an infrastructure and market for them. However, his proposal failed due to the fundamental differences between hardware components and software components, which have an immaterial nature and require different economics.

Component-based Software Engineering (CBSE) gained widespread attention during the 1990ths, when it became obvious that Object-Oriented Programming (OOP) [Mey97] had failed to effectively support reuse of software parts. Clemens Szyperski's book "Component Software" [SGM02] describes CBSE as the next step beyond OOP and was influential in establishing the terms and concepts of this area. CBSE gained attention from the software industry, where several component models, such as Microsoft's COM [Cor], Sun's EJB [EJB07], and the OMG's CCM [Obj06b], appeared to support the implementation of component-based systems.

Many promises of CBSE still have to be realised. The anticipated component marketplaces [SGM02, pp.18], where software developers shall purchase and exchange software components to rapidly build large systems, have not become reality as there is still limited reuse of software components across different companies. Compositional reasoning about the properties of systems based on component specifications is still an active research area that up-to-date has not produced industry relevant methods or tools [Gru07].

2.1.2 Software Components

In order to clarify the notion of a software component as understood in this thesis, this subsection provides a definition and explains some of the fundamental concepts regarding software components.

As an advancement of object-oriented technology, software components adhere to the same principles as software objects [Mey97]:

- **State:** A software object may have an encapsulated internal state, which describes the data stored in it.
- **Behaviour:** A software object can be used and its state can be manipulated via functions accessible by clients.
- **Identity:** A software object has a unique identity regardless of its internal state.

Software components extend these principles and shift emphasis to the specification instead of the implementation [CD01, pp.3]. Components have explicitly declared provided and required interfaces. Clients can access components only through their provided interfaces. Component can access other components only through their own required interfaces. There is a clear separation between the specification of a software component via its interfaces, and its implementation via code. With the publicly declared interfaces, it is possible to easily replace a component implementation by another one, which complies to the same component specification (i.e., it implements and requires the same interfaces).

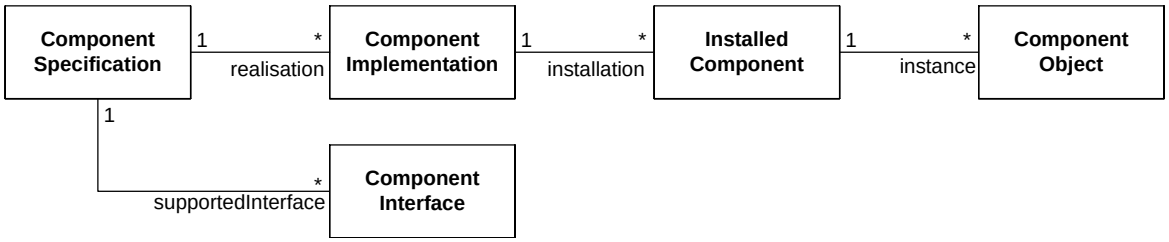


Figure 2.1: Forms of Software Components [CD01]

The notion of a software component is often confused in the literature, because of the different forms a component can have. Cheesman et al. [CD01, pp.4-7] distinguish between four different forms (Fig. 2.1). The *component specification* consists of a set of component interfaces, which declare the services provided by the component to clients and the services required by the component from its environment. Component developers can write different *component implementations* for the same component specification. This is a piece of code that implements the functionality specified in the provided interfaces by using services from the required interfaces.

A single component implementation can lead to multiple *installed components*. An installed component is assembled with other components and deployed on a platform. The component acquires its state only at runtime, when it is referred to as a *component object*. There may be multiple component objects for the same installed component resulting from multiple processes running in parallel. Each component object has a unique identity.

As an example for a large software component, consider a text editor application [CD01, pp.6-7]. Its component specification could be a textual description of its interfaces on a piece of paper at the developers' company. The developers have coded the text editor and compiled it into an executable, which is the component im-

plementation. Users can install this executable on many different machines, thereby creating many different installed components. On a single machine, a user can start the text editor twice, for example to edit two different files in parallel. This creates two different component objects of the same installed component.

The often cited definition of a software component by Szyperski [SGM02] from the 1996 workshop on component-oriented programming is often misinterpreted in the literature as it uses the term 'software component' synonymously with the term 'component specification'. It does not refer to the other forms of a software component described above. Nevertheless, the definition by Szyperski is also the one used in this thesis:

Definition 1 Software Component [SGM02, p.41]

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

As a *unit of composition*, a software component needs to be connectable to other components as a single entity. To enable composition, it has *contractually specified interfaces*, which refers to provided services of the component. According to Szyperski [SGM02, p.53], they should be specified after the design-by-contract principle [Mey92]. This implies that there is a pre-condition and a post-condition specified for each provided service. The contract states that if a client fulfills the pre-condition of a service, then the component guarantees its post-condition. *Explicit context dependencies* refers to required interfaces, which declare services of other components invoked by the component when executing its implementation. It can also refer to a certain platform standard the component requires to be deployed on [SGM02, p.44].

Besides this technical part, the definition's second sentence refers to the market-related part of a software component [SGM02, p.41]. *Independent Deployment* refers to the fact that a software component needs to be self-contained and sufficiently separated from other components so that it remains replaceable. Composition is carried out by *third parties*, which refers to the separation between component developers and component assemblers, which also requires an exact definition of the provided and required interfaces.

There are several other component properties not explicitly included in Szyperski's definition. Besides a list of signatures for the provided or required services, a component *interface* may also include a protocol specification that constrains valid call sequences of the services [BJPW99], which is a special case of the design-by-

contract principle. For example, a component for manipulating files may require clients to first call a service 'OpenFile', before it allows editing the file by calling other services.

A component *implementation* may be black-box or white-box, which refers to the visibility of the implementation to its clients [SGM02, p.40]. Black-box component implementations do no expose code to the clients thereby adhering to the principle of information hiding. As black-box components are usually delivered as binary code or bytecode, it is possible to retrieve additional information about their implementation using tools analysing these artefacts.

White-box component implementations allow clients to view and also edit the code. In general, it is desirable to have black-box components to ensure full replaceability. If clients know the code of a white-box component they could rely on certain implementation details of the component, which would disallow replacement of the component by another implementation.

There are also mixed forms of grey-box components, which reveal only parts of their implementations, and glass-box components, which let users only view the code but forbid editing. For performance predictions, black box component specifications are not sufficient, because they do not contain information about resource usage of component services. This thesis will introduce a modelling language to describe grey box views of software components, which includes resource demands and the order of calls to required services.

Components should only be usable via delegation, but not by *inheritance*. This is essentially a white-box adaptation technique, because it makes the inheriting object dependent on its parent [Reu01b]. The developer of the inheriting object needs to know specifics of the base class and might rely on its implementation details to avoid mistakes while overriding methods. Furthermore, the inherited classes can access the state of their base class. Therefore, this inheritance is considered a white-box adaptation technique. It also leads to the 'fragile base class' problem [SGM02]: if a base class is changed, it could require all inheriting children to recompile. Therefore, it is not desirable to use inheritance to define new components.

Component developers can create component implementations using any programming paradigm as long as they provide the functionality defined in its provided interfaces. A component can for example be written using object-oriented techniques, or using functional programming techniques. It is furthermore possible to declare component assemblies as new components by defining a new component specification for the component assembly ('composite component').

The *granularity* of a component, i.e., the amount of functionality bundled in a component, is variable [SGM02, p.45-46]. It mainly influences the context dependencies of a component. If a component developer aims for maximum reuse, as often done in object-oriented programming, everything is excluded from the component except its prime functionality. However, this is problematic as it leads to a substantial increase of the component's context dependencies, which makes it hard to use the component, because it requires to set-up a large amount of additional components. Szyperski refers to the phenomenon as "Maximum reuse limits use", and recommends to find a balance between component leanness (for maximum reuse) and component robustness (for maximum use).

Components are different from software modules [Par72], which in contrast are not contractually specified and do not contain explicit required interfaces, which limits their replaceability. Other than abstract data types (ADT) [LZ74], components describe their dependencies explicitly.

One aim of this thesis is to provide a modelling language for performance specification of component implementations. These specifications are highly parameterisable for different contextual influences to software components (for example user inputs or the deployment platform), which alter their performance properties and cannot be determined by the component developer during specification. Such specifications are an important prerequisite to enable compositional reasoning on the properties of a component-based system at design time.

2.1.3 Software Architecture

Assembling components and deploying them in an execution environment yields a software architecture. The methods and models proposed in this thesis aim at improving the design of software architectures, by enabling software architects to assess different design alternatives quantitatively with respect to performance. Therefore, the following provides some basic concept and terms centering around software architectures.

A definition of the term *software architecture* is given by the IEEE standard 1471-2000:

Definition 2 Software Architecture [IEE00]

The fundamental organisation of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.

Cheesman et al. [CD01] provide a more refined view of a software architecture composed out of software components and distinguish between a system architecture and a component architecture. The *system architecture* is the structure of parts that together make up a complete software system, which includes the responsibilities of these parts, their interconnections, and possibly also the appropriate technology. It may consist of multiple architectural layers, for example in Java EE [Sun] the user interface (e.g. implemented with Java Server Pages), the user dialogs holding the state of user interaction (e.g., implemented with Java Beans), the system services representing the business logic (e.g., J2EE session beans), and business services ensuring persistency (e.g., J2EE entity beans).

Included in this system architecture is the *component architecture*, which refers only to the set of application-level (excluding user interfaces and persistency) components, their structural relationships and their behavioural dependencies. In the former example this would refer only to the server part of the application. The component architecture is rather a logical concept independent from technical realisation. It includes the bindings between components called connectors, and is often expressed in UML component diagrams. When the term software architecture is used in this thesis, it usually refers to the component architecture, and not the system architecture.

Multiple views are used to describe different aspects of a software architecture [CBB+03]:

- **Static View:** shows the software components and their bindings, for example expressed in a UML component diagram.
- **Dynamic View:** shows the interaction of components in specific use cases, for example expressed in UML sequence diagrams.
- **Deployment View:** shows the mapping of components to hardware resources, for example expressed in a UML deployment diagram.

Having an explicitly documented software architecture instead of just source code bears several *advantages* [CBB+03]. It bridges the gap between requirements and code and divides a system into a limited set of manageable components. It can be the basis for discussions between different stakeholders of a software system. During design of a software architecture, reusing existing components can be considered systematically. A software architecture also simplifies project management, planning, cost estimation, division of work and might enable offshore development.

Finally, a carefully designed software architecture can be the basis for assessing the quality attributes of the final software system before implementation.

For the last point, there are qualitative and quantitative methods for assessing software architecture quality. Qualitative methods, such as SAAM [KBWA94] or ATAM [KKC00], require developers to define critical scenarios of a software architecture and assess them in group meetings. The results are for example critical points in the architecture or a list of required changes. Quantitative methods require developers to build formal models for software architectures and analyse them via mathematical methods or simulation. The results are for example the expected mean response time of a use case or the mean time to failure of a software component. The method introduced in this thesis is a quantitative method.

2.1.4 Component-based Development Process

Component-based software development follows a different process than classical procedural or object-oriented development [SGM02]. The task of developing software artefacts is split between the role of the component developer, who develops individual components, and the software architect, who assembles those components to form an application.

Cheesman and Daniels [CD01] have proposed a component-based development process model based on the Rational Unified Process (RUP). It describes the specification and implementation of a component-based software system. The model focusses on the *development process* that is concerned with creating a working system from requirements and neglects the concurrent *management process* that is concerned with time planning and controlling.

The Palladio Component Model described in the Chapter 3 and 4 of this thesis explicitly targets this process model. Because the model does not include the analysis of extra-functional properties, Chapter 3.1 will introduce an according extension to the model.

Fig. 2.2 illustrates the original main process by Cheesman et al. Each box represents a workflow. The thick arrows between boxes represent a change of activity, while the thin arrows characterise the flow of artefacts between the workflows. The workflows do not have to be traversed linearly (i.e., no waterfall model). Backward steps into former workflows are allowed. The model also allows an incremental or iterative development based on prototypes. The included workflows are:

- **Requirements:** The business requirements coming from customers are for-

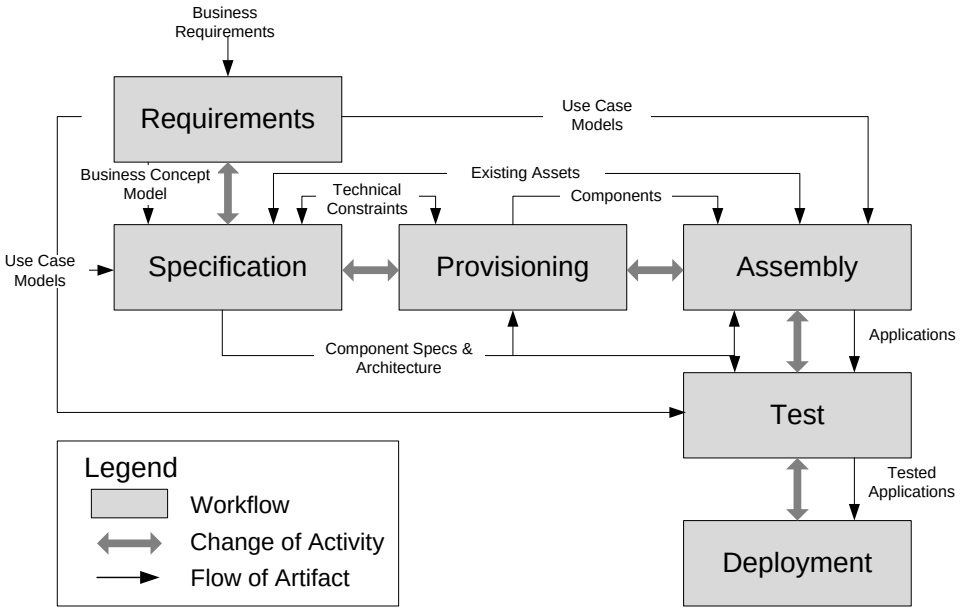


Figure 2.2: Component-Based Development Process [CD01]

malised and analysed during this workflow. It produces a business concept model and a use case model. The former is a conceptual model of the business domain and creates a common vocabulary between customers and developers. The latter describes the interaction between users (or other external actors) with the system. It establishes the system boundaries and a set of use cases that shall fulfill the functional requirements.

- **Specification:** During specification, the component-based software architecture is designed. The business concept model and the use case model are input from the requirements to this workflow. Additionally, technical constraints, which might have been revealed during provisioning, can be input to the specification workflow after initial iterations of the process model. Software architects identify components and specify them and their interactions during specification. They usually interact with component developers during this workflow or rely on existing component specifications. The output artefacts of this workflow are complete component specifications and the component architecture.

- **Provisioning:** Compared to classical development processes, the provisioning workflow resembles the classical implementation workflow. However, one of

the assets of component-based development is reuse, i.e., the incorporation of components developed by third parties. During the provisioning workflow 'make-or-buy' decisions are made for individual components. Components that cannot be purchased from third-parties have to be implemented according to the specifications from the corresponding workflow. Consequently, the provisioning workflow receives the component specifications and architecture as well as technical constraints as inputs. The output of this workflow are implemented software components.

- **Assembly:** Components from the provisioning workflow are used in the assembly workflow. Additionally, this workflow builds up on the component architecture and the use case model. The components are assembled according to the assembly model during this workflow. This might involve configuring them for specific component containers or frameworks. Furthermore, for integrating legacy components, it might be necessary to write adapters to bridge mismatching interfaces. Once all components are assembled, the complete application code is the output of this workflow.

- **Test:** The complete component-based application is tested according to the use case models in this workflow in a test environment. Once the functional properties have been tested in the test environment, the application is ready for deployment in the actual customer environment.

- **Deployment:** During deployment, the tested application is installed in its actual customer environment. The term deployment is also used to denote the process of putting components into component containers, but here the term refers to a broader task. Besides the installation, it might be necessary to adopt the resource environment at the customer's facilities or to instruct future users of the system with the new functionality.

2.2 Model-Driven Software Development

Many of the concepts and abstractions for component-based performance engineering developed for this thesis have been encoded into meta-models. Instances of these meta-models describe component-based software architectures and their performance properties. Therefore, the following subsection explains the basic terms and concepts of model-driven software development (MDSD) (including model,

meta-model, model transformation, etc.), which are necessary to understand the following chapters.

2.2.1 Introduction

MDSD aims at describing software systems at higher abstraction levels, which shall be more suitable to manage the increasing complexity in modern software applications. This approach is in line with former developments in computer science, which introduced assembler languages to raise the abstraction level from machine languages and then higher programming languages to raise the abstraction level from assembler languages. MDSD tries to shift the focus of software development from code written in higher programming languages to models. Model transformations shall generate code from these models, just as compilers transform high-level code to machine code.

To support MDSD and create an industry standard of its core concepts, the OMG started the Model-Driven Architecture (MDA) initiative in 2001 [KWB03]. Although the MDA became popular, there are still other approaches to MDSD, such as generative programming [CE00], Microsoft's Software Factories initiative [GSCK04], or product line engineering [CN02]. Like these approaches the MDA can be seen as a special flavor of MDSD. In this thesis, MDA concepts such as platform-independent and platform-specific models do not play a major role. Therefore, the following description of the main concepts follows a more general approach as in [SVC06].

2.2.2 Modelling Core Concepts

Stahl et al. [SVC06] describe the fundamental concepts of modelling and the relationships between these concepts (Fig. 2.3). Modelling is always conducted within a certain domain and involves a meta-model to create models. The following describes the different terms in Fig. 2.3 in detail. It uses counterparts from the MDA initiative and from the modelling approach underlying the later introduced Palladio Component Model to provide examples for the terms.

Domain Any modelling approach is tied to a specific *domain*.

Definition 3 Domain [SVC06]
A domain is a bounded field of interest or knowledge.

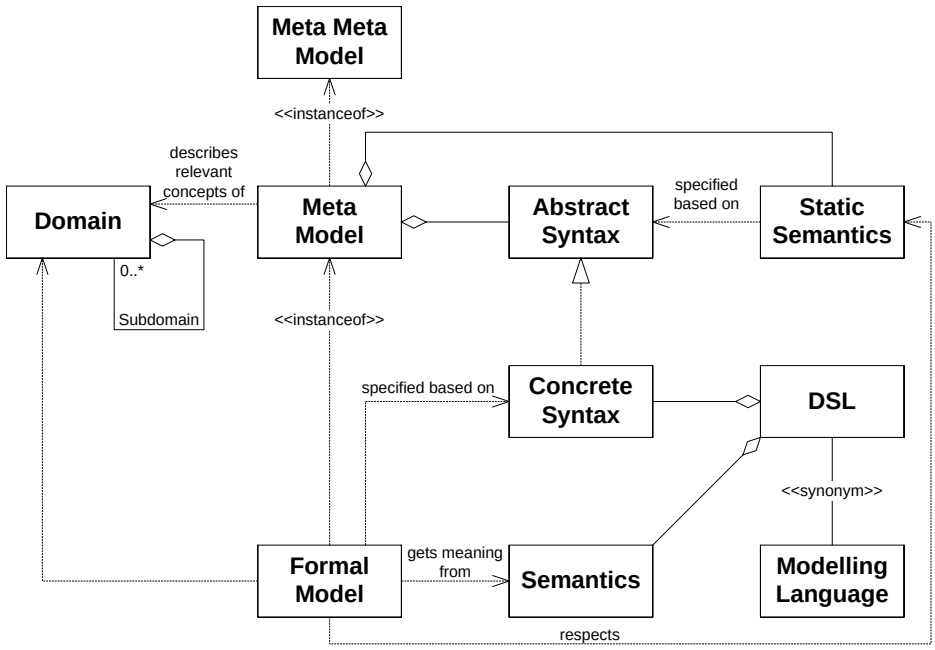


Figure 2.3: Modelling and DSLs [SVC06, p.56]

There may be different kinds of domains, for example professional domains or technical domains. An example for a professional domain would be ”banking”, which includes concepts like customer, account, accounting entry, and balance. An example for a technical domain would be ”Enterprise Java” with concepts like session bean, deployment descriptor and application server. There may also be subdomains describing parts of a domain, which are suited for an own modelling language.

The MDA initiative uses a technical domain, which is rather coarse-grained, it is ”software development”. In this thesis and for the PCM, the domain is also technical, but it is more focussed than in the MDA and can be called ”component-based performance engineering”. Later, this thesis shows the division of the PCM into several modelling languages, which target subdomains of specific developer roles in component-based performance engineering, such as the component developer, or the software architect.

Meta-Model A *meta-model* structures the domain and formalises its concepts. It can be considered as an ontology for the domain.

The meta-model defines which models developers can create, in other words it

Definition 4 Meta-model [met07]
A meta-model is a precise definition of the constructs and rules needed for creating semantic models.

is the grammar for the language of models in a domain. The models are instances of the meta-model. A meta-model is needed for automation, such as model transformation or code generation.

In MDA, the main meta-model for software development is the UML2 meta-model. There are also several profiles for UML, which include meta-models for specific domains and extend UML2 meta-model. Examples are profiles for CORBA systems, performance modelling, or software testing. In this thesis, the meta-model is the Palladio Component Model, which is introduced in Chapter 3 and extended in Chapter 4. It formalises concepts of the domain component-based performance engineering.

Meta-Meta-Model The meta-model itself has a meta-model, which is called meta-meta-model.

Definition 5 Meta-Meta-Model [SVC06]
A meta-meta-model defines the concepts available for meta-modelling.

In general the term meta is relative, as there can be arbitrary many meta-levels. In the MDA, the meta-meta-model is called Meta Object Facility (MOF) [Obj06c] and the UML2 meta-model is defined in MOF. Instead, the PCM's meta-meta-model is Ecore, an implementation of a subset of MOF called Essential MOF (EMOF). Ecore is part of the Eclipse Modeling Framework (EMF) [Eclb]. This meta-meta-model has been chosen because of its limited complexity and because of extensive tool and community support. Whenever meta-model parts of the PCM are shown in the remainder of this thesis, they are Ecore instances. Ecore allows the definition of packages, classes, attributes, operations, data types, references etc.

Abstract and Concrete Syntax A meta-model compasses an abstract syntax that specifies the language's structure.

For example in Java, the abstract syntax tree of a Java program would be a representation of a word of the Java language in abstract syntax. In XML, the DOM tree created by parsing an XML document is the abstract syntax. Usually, developers do not use the abstract syntax directly to specify words of the language. Instead, they

Definition 6 Abstract Syntax [FOL08]

Abstract syntax is a representation of data which is independent of machine-oriented structures and encodings and also of the physical representation of the data (called "concrete syntax" in the case of compilation).

Definition 7 Concrete Syntax [FOL08]

The concrete syntax of a language including all the features visible in the source program such as parentheses and delimiters. The concrete syntax is used when parsing the program or other input, during which it is usually converted into some kind of abstract syntax tree.

use a more convenient, so-called concrete syntax. This is a notation for developers to create models that are instances of meta-model. In Java, the concrete syntax is the usual program code, which is accepted by the parser. In XML, the tags and values used in an XML document form the concrete syntax. It is possible to have multiple concrete syntaxes for the same abstract syntax, for example a textual and a graphical notation.

In the MDA, the abstract syntax of the UML2 meta-model is a MOF instance diagram. As a concrete syntax, usually the diagrams (e.g., class diagrams, sequence diagrams, etc.) defined in the UML standard are used. However, each UML diagram could also be expressed in abstract syntax using MOF. In this thesis, the abstract syntax of PCM instances can be given using Ecore. As concrete syntax, there are tree-like representations of PCM instances via editors generated with EMF and also graphical representations from graphical editors implemented for PCM instances using the Graphical Modeling Framework (GMF) [Eclc].

Static Semantics Besides an abstract syntax, a meta-model also includes a definition of static semantics.

Definition 8 Static Semantics [Mey90]

Static semantics describe structural constraints that cannot be adequately captured by syntax descriptions.

For example, in programming languages, a typical example for such a well-formedness constraint is that each variable has to be declared before using it. Notice, that the static semantics do not refer to the meaning of the meta-model classes, but only to the well-formedness of instances.

In MDA, developers express static semantics of meta-models using the Object Constraint Language (OCL). This is a declarative language, which refers to the abstract syntax of the UML2 meta-model. In the PCM, also OCL expressions are used to specify the static semantics. These expressions are defined on the Ecore model.

Definition 9 Semantics [SVC06]
Semantics give meaning to the modellable elements and indicate the entities they represent.

Semantics Semantics are either intuitively clear or well-documented (for example in natural language). Semantics can also be given by describing a mapping to another language with formal semantics. This can for example be applied for mathematically related languages.

In the MDA, the semantics of UML2 are only given informally in natural language in the UML specification. In this thesis, the semantics of the PCM are first described textually in natural languages. Later, in Chapter 4.4, a mapping from PCM instances to queueing Petri nets (QPNs) is defined to capture the performance-related aspects of the PCM formally. For QPNs in turn, mappings to Colored Petri Nets, and finally to simpler mathematical structures like stochastic processes have been defined in the literature (also see Appendix B.2).

Domain-Specific Language (DSL) Developers use DSLs to model the key elements of a domain and formally express them.

Definition 10 Domain-Specific Language [vDKV00]
A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

A DSL consists of a meta-model, a concrete syntax, and dynamic semantics. The term 'modelling-language' is often used synonymously with DSL. A DSL is usually accompanied by a (graphical) editor, which supports entering model instances for the meta-models in concrete syntax [SVC06].

In the MDA, the UML2 meta-model together with the diagram notation and the semantics described informally in the UML specification forms a DSL for software development. There are many UML modelling tools, which support drawing UML diagrams, therefore they support the concrete UML syntax.

For the PCM, there is the idea to split the meta-model into four distinctive DSLs, each one for a different developer role, such as component developer or software architect. For each of these DSLs, a concrete syntax in form of a graphical representation of the formal models is available. There are individual graphical editors for each modelling language. This allows the different developer roles to work independently from each other.

Model Models are instances of meta-models and are the artefacts developer create to build an abstraction of a software system for some specific goal (e.g., documentation, communication, formal verification).

Definition 11 Model [Sta73]
A formal representation of entities and relationships in the real world (abstraction) with a certain correspondence (isomorphism) for a certain purpose (pragmatics).

A model needs a DSL and is specified using concrete syntax. For example, a Java program is an instance of the Java language, thus a model for the Java grammar.

In the MDA, models are instances of the UML2 meta-model (UML2 models) and describe structural or behavioural aspects of software systems for documentation and communication purposes. Sometimes these models are also subject to model transformations, which generate program code or other models. In the PCM, models are PCM instances created via the graphical editors for the DSLs. They describe structural and behavioural aspects of component-based software architecture with a special focus on their performance properties. PCM instances can be transformed into general performance prediction models, which do not include the notion of a software component.

2.2.3 Transformations

Model transformations process models for different purposes [SVC06]. They may generate source code from models or transform them into other models for specific analysis methods on the model level. The transformation rules can only refer to the meta-model constructs.There are generally model-to-code and model-to-model transformations, although model-to-code transformations can be viewed as a special form of model-to-model transformations. Model-to-code transformations however usually do not need a target meta-model, because they directly create text files

in a specific syntax. These model-to-code transformations are however irrelevant for this thesis.

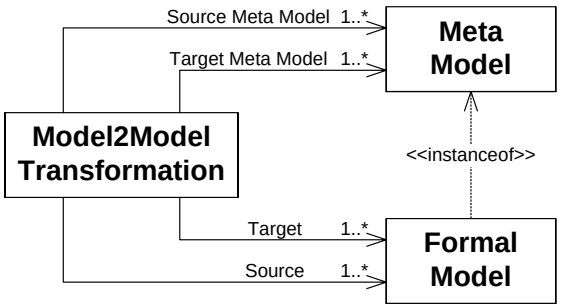


Figure 2.4: Model Transformations [SVC06, p.60]

Model-to-model transformations (Fig. 2.4) create a new model from a source model. Normally, the target model is based on a different meta-model than the source model, otherwise the transformation is referred to as an in-place transformation. A model-to-model transformations describes a mapping from source meta-model elements to target meta-model elements.

In the MDA, Query/View/Transformations (QVT) [Obj06d] is the standard for model transformations standardised by the OMG. It is divided into a declarative part (QVT Relations) and an imperative part (QVT Operational Mappings). While several engines for QVT Operational Mappings have been created, there are only very few implementations of QVT Relations, as the standard is still quite new.

Besides QVT, there are many other model transformation languages, which have been classified by Czarnecki and Helsen [CH06]. For example, there are visitor-based, template-based, or graph-grammar based approaches.

Chapter 6 of this thesis describes two model-to-model transformations to map PCM instances to different performance models. Due to the immaturity of the QVT implementations and the complexity of the transformations, which involve mathematical operations to bridge semantic differences between source and target meta-models, the transformations in this thesis have been implemented using the Java programming languages. Using special model visitors provided by EMF, these transformations process PCM instances and create new target models conforming to meta-models of the performance model, which are like the PCM implemented in Ecore.

2.3 Performance Engineering

The meta-models developed in this thesis target at modelling the performance properties of component-based systems. They shall help component developers to specify the performance of their implementations, so that software architects can build a performance model for the whole system, which enables performance predictions and the assessment of the performance of different design alternatives. The area of Performance Engineering [Jai91] has a long history in computer science and offers different methods to evaluate the performance of computer systems.

This subsection gives a short overview on some central concepts in performance engineering to make the reader familiar with the context of the methods and models proposed in this thesis. Chapter 2.3.1 briefly describes the most important metrics for performance and explains the three principal methods for performance evaluation. Chapter 2.3.2 surveys some of the classical performance models, such as queueing networks and stochastic Petri nets. While these classical models center around modelling hardware resources of a system, the Software Performance Engineering (SPE) approach initiated by Smith [Smi90] emphasises the influence of software on performance properties and is discussed in Chapter 2.3.3. Recently, several performance meta-models have been proposed in the literature, as it is the goal to create an ontology for performance modelling [Cor05]. Chapter 2.3.4 briefly describes these models, as they are loosely related to the Palladio Component Model.

2.3.1 Introduction

Performance Metrics In computer science, the term 'performance' is often used as a collective term to characterise the timing behaviour and resource efficiency of a hardware/software system. Performance properties of a system can be quantified with different performance metrics. The most important ones are response time, throughput, and resource utilisation [Jai91]. Many other performance metrics are simply derivates from these three main metrics.

The *response time* of a system refers to the time the system needs to process requests from the user's point of view. A low response time ensures that users can access the functionality of a system without delays interrupting their workflows. Response time can be defined differently depending on the start- and end-point of a request. Sometimes it is the time between issuing a request and receiving the full answer. Other approaches define the response time as the time between issuing a request and the system starting to answer the request. This is a user-oriented met-

ric, and some systems guarantee certain maximum response times with a so-called service level agreement (SLA).

The *throughput* of a system refers to the rate at which a system handles requests. It is usually measured in tasks per time unit [ISO03]. Besides throughput for the whole system, also the throughput of single resources is sometimes interesting for performance analysis. Many performance evaluation studies aim at determining the system capacity (i.e., the maximum throughput of a system for a given maximum response time). This metric is interesting for system developers, who desire a large throughput to serve as many customers as possible.

The *resource utilisation* refers to the ratio of busy time of a resource by the total elapsed time of a measurement period. For example, if a CPU processes requests for 68 seconds during a 100 second measurement period, its resource utilisation is 68 percent. The resource, which exhibits the highest utilisation is called the 'bottleneck' of the system and is usually the starting point when adapting a system for increasing its performance. This metric is mainly interesting for performance analysts and system admins, who want to ensure a balanced use of the available resources. If resources are fully utilised over a longer time period, the response time of the system may degrade. On the other hand, if the system resources are constantly underused, their potential processing power is essentially wasted.

Performance Evaluation Methods To assess the performance metrics explained above, there are three different classes of methods: analytical modelling, simulations, and measurements [Jai91].

Analytical modelling involves creating performance models (such as queuing networks, stochastic Petri nets, or stochastic process algebras, cf. Chapter 2.3.2) and deriving performance metrics by solving the underlying Markov chains mathematically using exact methods or heuristics. Performance analysts can determine the parameters for such models (e.g., the service demand times for different resource, the number of users concurrently present in the system, or the probabilities of accessing specific resources) via measurements on existing system or via estimations based on experience with similar systems. Measurements can also be performed on prototypes of a system, so that the values can be determined during early development stages.

The advantages of analytical models are quick creation compared to simulations, fast analysis, and low costs [Jai91]. Using analytical models, performance analysts can easily investigate trade-offs, for example for using different resources. The dis-

advantages of analytical models are the strong assumptions underlying these models, which often do not hold in realistic systems leading to inaccurate predictions.

Simulation also involves creating a model of a system, which is often based on queueing networks. There are several simulation frameworks available for different programming languages, which enable creation of simulation models. Other than using mathematical techniques to derive performance metrics, simulation involves executing the models repeatedly to imitate the performance properties of the modelled system. This is not performed in real-time, but the simulation simply adds the time consumed by a request on different resources and calculates waiting delays due to resource contention with other requests (event-driven execution), which is usually much quicker than actually executing the real system.

The advantages of simulations are that they can be conducted during any development stage, and that they do not impose any assumptions on the modelled system behaviour as the mathematical models do. Simulation models can be arbitrarily detailed, whereas analytical models often suffer from state space explosion in non-trivial cases. The disadvantages of simulation are the potentially high costs to build a detailed simulation model and the longer execution times compared to analytical solutions to get sufficiently accurate performance metrics [Jai91].

Measurements require executing the system under analysis and monitoring its performance properties using profiling tools or benchmarks. This requires an implementation of the system or at least a prototype [BCC+05]. Thus, measurements usually are only conducted during late development stages. Besides a system implementation, performance analysts have to reproduce the typical workload of using the system and provide a realistic target hardware environment, so that the measured metrics are representative for the actual system performance.

As advantages, measurements are usually very accurate and more convincing for management than the results of analytical models or simulations. As disadvantages, measurements are costly because measurement facilities have to be set up, the workload has to be reproduced, and hardware has to be procured. Performance analysts can conduct measurements only during late development cycles, when discovering performance problems might require expensive re-designs or re-implementations if the architecture is the root of these problems. Trade-off analyses are hard to perform with measurements, because different hardware would have to be purchased and set up, which is usually not affordable or desirable.

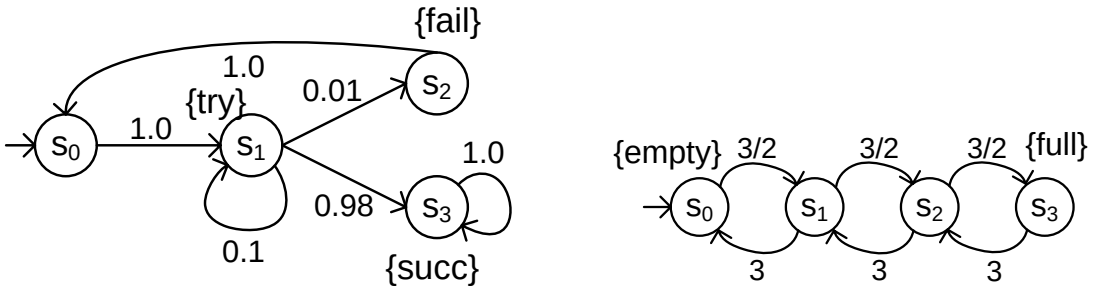
In this thesis, both analytical models and simulation are used to derive performance metrics from models. In Chapter 7, measurement on prototypical implemen-

tations of component-based systems are used to validate the predictions made with analytical and simulative predictions.

2.3.2 Performance Modelling

The following briefly overviews some of the most important classical performance models. For brevity, this section abstains from formal definitions, but uses simple examples to give the reader an initial idea of the formalisms. More details about these formalisms can for example be found in [BH07]. For these models, analytical and simulative solvers are available to derive performance metrics.

Markov Chains A Markov chain is a discrete-time stochastic process with the Markov property [Tri01]. The Markov property means that choosing the next state in the chain depends only on the current state and not on any previous states. Markov chains have been successfully used for performance evaluation of computer systems. Most of the other performance modelling formalisms can be transformed into Markov chains using state space generation techniques [BH07].



(a) DTMC (b) CTMC

Figure 2.5: Markov Chains

There are discrete-time Markov chains (DTMC), where the state space and the time for changing states is discrete, and continuous-time Markov chains (CTMC), where the state space is discrete, but the transition from a given state to another can occur at any instant of time [Tri01]. Markov chains can be visualised as directed graphs were the vertices represent states and the edges represent transitions and are labelled with transition probabilities (DTMC, Fig. 2.5(a)) or transition rates (CTMC, Fig. 2.5(b)). However, for calculations, Markov chains are usually represented by a transition matrix (DTMC) or a so-called infinitesimal generator matrix (CTMC). In the examples depicted above, the DTMC represents the behaviour of a user logging

into a system, and the states of the CTMC represent the current length of a queue in front of a processor.

There are different solution techniques for Markov chains depending on the type of the chain. If a Markov chain has an absorbing state that cannot be left anymore once it is reached (e.g., s_3 in Fig. 2.5(a)), transient analysis techniques are used. If a Markov chain only contains recurrent states, i.e., states, where the process will return to eventually with a probability of 1 (e.g. as in Fig. 2.5(b)), steady state probabilities can be calculated. These express the probability of being in a certain state if the process runs infinitely long. From these steady state probabilities other performance metrics can be derived.

Calculating steady state probabilities involves solving linear equation systems. Because Markov chains for realistic system are usually very large, there are several heuristic solutions techniques, which are able to provide approximate solutions the large linear equations systems, because exact solutions cannot be computed efficiently. Because of their low abstraction level, Markov chains are seldom specified manually in performance engineering. Instead, high level formalisms, such as queueing networks or stochastic Petri nets, are used for specification and corresponding Markov chains are generated from them using tools.

In the context of this thesis, Chapter 2.5.3 briefly discusses so-called Markov usage models. Furthermore, in Chapter 6.3, a transformation from the PCM into a special kind of semi-Markov chain will be described.

Queueing Networks Queueing theory has been applied to computer systems for performance engineering already since the 1970th [Smi01]. In fact, most recent performance prediction methods use queueing networks (QN) as performance analysis model [BDIS04]. A QN consists of a number of service centers, which usually represent the hardware resources in a computer system. Each service center (also called queueing system) consists of a server and a queue (cf. Fig. 2.6).

Service centers have different attributes, which are usually expressed in Kendall's notation [LZGS84]. This notation has the form $A/B/S/K/N/Disc$, where

- A is the inter-arrival time distribution for requests to a server
- B is the service time distribution (i.e., for the processing time of each request)
- S is the number of servers
- K is the system capacity

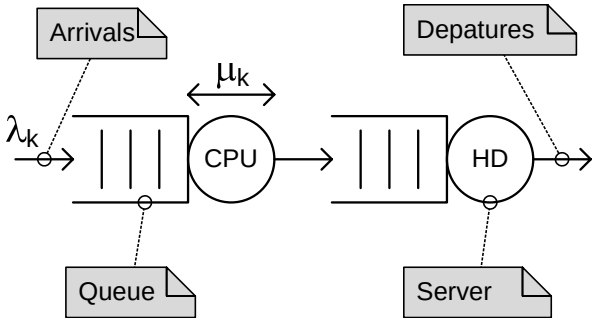


Figure 2.6: Queueing Network

- N is the calling population
- $Disc$ is the service discipline (e.g., FCFS, Round Robin, etc., cf. Appendix B.3).

There are also standard abbreviation for the time distributions (for A and B), such as M for a Markovian (i.e., exponential) distribution, G for a general distribution, or PH for a phase-type distribution. Often, approaches simply use the short notation $A/B/S$ to characterise the service centers. Important kinds of service centers, which allow analytical calculations of different mean-value performance metrics, are for example $M/M/1$, $M/M/m$, $M/M/\infty$, or $M/G/1$.

There are QNs with closed workloads (i.e., a fixed number of request circulate in the system), and open workloads (i.e., the number of requests is not fixed and requests depart from the system after finishing execution). For a special, restricted class of QNs, so-called product form QNs, it is possible to calculate performance indices without generating and solving the associated Markov chain. Therefore, efficient solution algorithms, e.g., the Convolution Algorithm and Mean-Value Analysis (MVA), have been developed for these kinds of QNs [BH07].

In general, many kinds of QNs with efficient mathematical solutions are based on hard assumptions about the modelled systems [Jai91]. Such assumptions for example include exponentially distributed service times or an infinite user population. These assumptions do not hold for realistic systems. Queueing models with weaker assumptions are usually mathematically intractable and can only be simulated to derive approximated performance indices.

The PCM usage model introduced in Chapter 4.2 models the user population in a system like QNs using open or closed workloads. The PCM resource model uses $G/G/1$ service centers that require simulations for multi-user cases. In Chapter 6.4, a model transformation maps PCM instances to so-called Layered Queueing Networks (LQN), which include modelling software entities.

Stochastic Petri Nets One problem of QNs in terms of performance modelling is that they support concurrently interacting requests only poorly, because they do not provide special mechanisms for synchronisation [BH07]. Therefore, it is not possible to model such situations, which often occur in realistic systems. However, Petri nets offer simple mechanisms to synchronise concurrent requests via shared places. As ordinary Petri nets do not contain any timing information needed for expressing execution times, several proposals have been made to augment Petri nets with timing annotations since 1980.

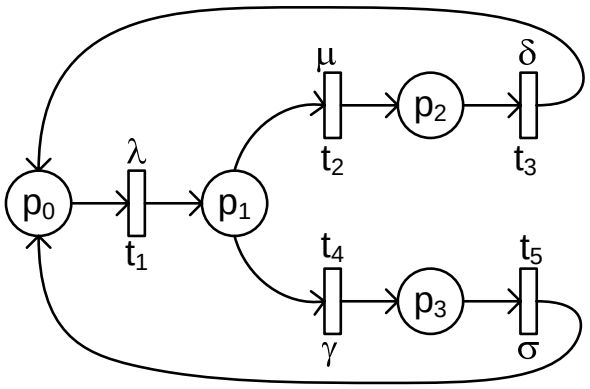


Figure 2.7: Stochastic Petri Net

A Petri net consists of a set of places (representing state) and transitions (representing activities). Places and transitions are connected via arcs (cf. Fig. 2.7). Places may contain tokens (representing requests), which can be moved through the nets by firing the transitions. The current distribution of tokens on the places of a Petri net is called marking. A formal definition of a Petri net is given in Appendix B.2.

Stochastic Petri nets (SPN) may contain *timed* transitions, which are annotated by a firing rate that specifies a parameter of the exponential distribution of the time span between subsequent firings of the transition. Thus, SPNs allow performance analysts to model the timing behaviour of a system. Generalised Stochastic Petri Nets (GSPN) additionally may contain *immediate* transitions, which are annotated by a firing weight that specified the probability that the transition will be fired, if multiple transitions are enabled at the same time. Using immediate transitions, performance analysts can model the behaviour of a system with stochastic means instead of being limited to deterministic behaviour.

To derive performance indices from SPNs, either analytical techniques or simulation are available. Analytical techniques construct the so-called reachability graph from a SPN, which is a labelled transition system, where each state represents a pos-

sible marking of the net. With the firing rates, this is a CTMC, which can be solved using Markov chain solution techniques. Instead, simulations execute an SPN, i.e., starting from an initial marking, they move tokens through the net by firing the transitions. After a simulation run reaches a predefined stop criteria, various performance indices can be derived from the simulations data.

SPNs have similar assumptions as Markov chains. They only allow exponentially distributed firing rates, other probability distribution have to be approximated using phase-type distributions. Another limitation of SPNs is the cumbersome modelling of service centers as in QNs, which are useful representations of hardware resources. Therefore, extensions such as Queueing Petri nets [Bau93] (QPN) have been introduced.

In the context of this thesis, PCM usage models and RDSEFF will be mapped to QPNs in Chapter 4.4 to specify their performance-related formal semantics.

Stochastic Process Algebra Another formalism for performance modelling is a Stochastic Process Algebra (SPA) [BH07]. SPAs contain explicit operators with formally defined semantics to compose different models to larger models. This is supported by QNs and SPNs only implicitly with no formal semantics. Using an SPA allows specifying multiple processes, which run in parallel and interact with each other, via simple composition. This is also advantageous to reuse existing process specification in different system models.

SPAs build on classical process algebras, such as Millner's Calculus of Communicating Systems (CCS) [Mil89] and Hoare's Communicating Sequential Processes (CSP) [Hoa85]. While these formalisms focus on assessing functional correctness of a system for example by showing the absence of deadlocks, SPAs extent them with stochastic and timing information to also allow deriving performance properties.

There are several Markovian SPAs, which feature exponential distributions for timing delays, such as TIPP [GHR92], EMPA [BG98], and PEPA [Hil96]. Some other calculi also incorporate general distributions for delays, such as MODEST [BDHK06], IGSMP [BG02] and GSMPA [BBG97]. However, the latter do not offer numerical solutions for deriving performance indices if general distributions are used and must be analysed using simulations.

In the context of this thesis, no SPAs are used, although the SREs introduced in Chapter 6.3 can be seen as a simple form of an SPA with generally distributed timing annotations. Happe [Hap08] is currently extending this formalism.

2.3.3 Software Performance Engineering

Because QNs and other performance models account for software performance only implicitly, Connie Smith initiated the Software Performance Engineering (SPE) approach during the 1980th [Smi90]. In QNs, software behaviour is usually condensed to the service demand of a request to a hardware resource, which is a single value per request. There is no direct representation of the software architecture and the control flow through the system, as QNs focus on hardware resources.

Besides the shift from hardware-centric models to mixed software/hardware models, Smith also postulates to conduct performance evaluation as early as possible during the life-cycle of a system [Smi90, Smi02]. In the software industry, todays prevalent approach to software development is to first ensure the functional properties of a system. Developers often care for the non-functional properties such as performance and reliability only during late development cycles when code artefacts are available for measurements. This so-called 'fix-it-later' approach is problematic, if performance problems are the result of a poorly designed architecture. When they are found after the system is already implemented, this might require expensive redesigns and coding.

To counter the 'fix-it-later' approach, the SPE methodology enables software developers to rapidly create simple models for analysing the performance properties of newly designed systems as early as possible, even if some details are still unknown. Steps of the method include (i) the definition of performance goals, (ii) the identification of key performance scenarios, (iii) the creation of a software execution model, (iv) the creation of a system execution model, and finally (v) performance analysis and revising the models.

Performance goals include performance-related requirements of a system and shall guide the creation of models. For example, a performance goal could be a maximum response time for a certain functionality, or a minimum number of users the system needs to handle concurrently without performance degradation. These performance requirements are also called service-level agreements (SLA). To keep the modelling effort low, SPE encourages only modelling performance-critical user scenarios, i.e., use cases that are likely to suffer from performance problems.

In SPE, software architects or performance analysts create a software execution model for each of the identified performance-critical user scenarios. This is an annotated control flow graph, which models the requests to different objects or components to carry out each step of the scenario. Additionally, the performance analyst annotates each node in the control flow graph with its demand for hardware

resources (e.g., a certain number of CPU instructions, or the number of hard disk accesses). Initially, this model can be coarse grain, as many details of the system might still be unknown. It can be refined during later development stages, for example if certain parts of the architecture have been implemented and can be measured.

The performance analyst also specifies a so-called overhead matrix, which includes timing values for the different kind of resource demands used in the annotated control flow graph. For example, it could specify the time to execute a CPU instruction or to access a hard disk. The matrix can be used to adjust specific performance values during performance analysis, for example to assess the impact of a faster CPU on the overall performance.

Multiple scenarios modelled as annotated control flow graphs can be combined to form a system execution model. From this fully specified model, tools can generate a QN, which can be solved with analytical techniques for single scenarios, or simulation for multiple concurrent scenarios. The SPE-ED tool accompanying the SPE methodology supports this [Smi02]. The resulting performance metrics are end-to-end response time for scenarios, throughput, and utilisation of different resources.

While the method proposed in this thesis is based on similar considerations as the SPE methodology (e.g., early design-time prediction, emphasis on software), there are also several differences between SPE and Palladio. SPE does not target component-based software systems, although applying the method on a component-based system is possible. However, the resulting SPE models are not reusable, as they lack the required parameterisation. Furthermore, these models may be on a higher abstraction level than the component architecture, therefore it is not possible to deduce individual component performance specification from them. Introducing a new component into a system design would require performance analysts to recreate the performance model from scratch.

2.3.4 Performance Meta-Models

Recently, researchers have proposed several meta-models, which include concepts of the SPE domain. They are specifically designed to enable and simplify model transformations. Among them are the UML SPT profile [Obj05b], the SPE-metamodel [SLC+05], the Core Scenario Model (CSM) [PW06], and KLAPER [GMS07b]. There is also a new UML profile in development called MARTE [Obj07a]. Cortellessa [Cor05] even asked, whether there will be a common software performance

ontology based on these meta-models, which would provide a standardised set of concepts of the domain. However, this is still subject to research.

Although these meta-models have different goals than the Palladio Component Model, their elements and concepts were influential in defining the PCM. The following briefly discusses the SPT profile, CSM, and KLAPER and their relation to the PCM.

UML SPT The UML SPT profile [Obj05b] is an extension to the UML, which enables developers to add performance related information to UML models. The aim is to reuse existing design documents and augment them with performance annotations. These annotated models shall be input for model transformations, which map them to performance models, such as QN, SPNs, or SPAs. Results from solving these models shall be feed back into the UML models, therefore the SPT profile includes corresponding annotations to save performance metrics from analysis tools.

The SPT meta-model for the performance domain is based on the General Resource Modeling (GRM) Framework, which distinguishes between different types of resources, such as active and passive resources. Software developers can specify scenarios consisting of several steps, which produce load onto the modelled resources. Furthermore, they may specify the workload of a scenario (e.g., the number of concurrent users). The SPT meta-model does not contain control flow constructs, as these shall be modelled with UML, for example by using activity diagrams or sequence diagrams.

This meta-model does not specifically target component-based system, and there is no support for creating models for individual software components. Marzolla [Mar04] has implemented a QN-based simulation for UML models annotated according to the SPT profile. Several other approaches use the profile for performance modelling [BDIS04].

CSM The Core Scenario Model (CSM) [PW06] is directly connected to the UML SPT profile. As UML designers may use different kinds of diagrams for expressing the performance properties of their systems (e.g., activity diagrams, sequence diagrams, collaboration diagrams, etc.), CSM aims at providing a common intermediate model as the target for mapping the different annotated UML diagrams. Model transformations to the performance domains (e.g., to QNs, SPNs, SPAs) then only have to be defined for the intermediate model instead of for each UML diagram.

Other than the SPT profile, CSM explicitly model control flow as sequences, al-

ternatives, loops, and forks. Furthermore, it adds acquire and release actions for passive resources and allows specifying message sizes for invocations, which is so far not possible with the SPT profile. While CSM has an entity 'Component', this does not model a software component in the sense of Szyperski, as it is instead a special passive resource.

Like the SPT profile, CSM does not target reusable models for individual components.

KLAPER The Kernel Language for Performance and Reliability Analysis (KLAPER) [GMS05, GMS07b] is another intermediate language to simplify model transformations. With such a language, the number of model transformations for N input models in the software design domain to M target models in the performance domain shall be reduced from $N * M$ to $N + M$. KLAPER targets component-based systems, but is not meant to be a specification language used by developers, but only by model transformation tools.

KLAPER includes a unified concept for software components and hardware resources (i.e., they are treated equally). This distorts Szyperski's component notion, as a hardware resource is usually not contractually specified and does not require other resources. However, KLAPER aims at enabling to integrate different component performance specification based on different notations, such as annotated UML models or OWL-S [MBH+04]. There are several model transformation from KLAPER to Markov chains or extended queueing networks, but the transformations from UML to KLAPER have not been finished so far. KLAPER has been extended for reconfigurable architectures [GMS07a].

While some of the concepts used in KLAPER also appear in the PCM, both approaches follow different directions. KLAPER shall only be used by model transformation tools, whereas the PCM shall be used an input language for different developer roles.

2.4 Component-Based Performance Engineering

2.4.1 Motivation

CBSE requires different performance modelling techniques than conventional SE in order to exploit the advantages of components. The following distinguishes between component-based systems with replaceable parts built by separated devel-

oper roles, and monolithic systems built by a single developer role. Monolithic systems may use other structuring mechanisms than componentry and be programmed with any programming paradigm just as components themselves.

Conventional software performance engineering techniques mostly focus on performance modelling and prediction for monolithic systems and require a single developer role (i.e., a performance analyst) to be able to model the whole system [Smi02]. While these techniques in principle also work with white-box component-based systems, they are limited for black-box component-based systems and cannot exploit the component paradigm for division of work and more accuracy.

For component-based systems, it is desirable that individual component developers specify the performance of their components and put these specifications into public repositories. Software architects can then compose these specifications isomorphically to the component implementations in the software architecture to conduct design-time performance predictions.

This has several potential benefits over conventional, system-wide performance modelling. The work for performance modelling is divided to different developer roles, who each contribute information from their domain. Component performance specifications can potentially be more accurate, because they may result from measuring an implementation instead of estimating execution times for designed, but yet unimplemented parts. Because of the intended reuse, components can be extensively tested and even performance-tuned. The experience from former uses of the component might refine the performance models. Furthermore, at least partial automatic model generation from existing component implementations is possible (cf. Chapter 5).

However, it is not easy for component developers to specify the performance of a software component, because they cannot make any assumptions on external influencing factors. A component performance specification therefore needs to be parametrisable for different contexts to become reusable for different software architects.

The potential higher accuracy of individual component performance models also has a drawback, as it might bloat these specification so much that the resulting system models become mathematically intractable for analysis methods. Also simulation techniques can only handle a certain amount of complexity. Thus, a performance specification language needs to find a good abstraction from precise component performance behaviour. It needs to be abstract enough to enable analysis methods on the one hand, but on the other hand allow accurate predictions.

To identify the right abstraction level, first the factors influencing the performance of software components need to be analysed.

2.4.2 Factors Influencing Performance of Software Components

The following lists several factors influencing the performance of a software component.

- **Implemented Algorithms:** Component developers can implement the functionality specified by an interface differently. For example, when implementing a component for sorting lists, one component developer could use the bubble sort algorithm, while another component developer could use the quick sort algorithm. The two components would exhibit different execution times running on the same resources and given the same inputs.
- **Service Parameters and Internal State:** Clients can invoke component services with different input parameters. The execution time of a service can change depending on the values of the input parameters. For example, if a component service for sorting lists is invoked with a short list, its execution time is shorter than if it would be called with a long list. Besides input parameters of provided services, components may also receive parameters as the result of calls to required services. The values of these parameters can also influence the execution time of a service. Furthermore, components can have an internal state from initialisation or former executions, which changes execution times.
- **Performance of Required Services:** When a component service invokes required services, their execution time adds up to its own time. Therefore, the overall execution time of a component service depends on the execution time of required services.
- **Resource Contention:** A software component typically does not execute as a single process in isolation on a given platform. Usually, processes of other applications are running in parallel or the component itself is accessed concurrently by multiple threads. Concurrently executed software processes produce contention on the resource of the underlying platform. For example, if multiple processes are executed on a single-core processor, the operating system scheduler queues requests and manages the order of execution. Therefore, the induced waiting times for accessing limited resources add up to the execution time of a software component.

- **Deployment Platform:** A software component can be deployed to different platforms. A deployment platform may include several software layers (e.g., component container, virtual machine, operating system, etc.) and hardware (e.g., processor, storage device, network, etc.). Depending on the speed of hardware resources, the scheduling disciplines of operating systems, and features of the middleware, the execution time of a software component can change significantly.

2.4.3 Requirements for Component Performance Specification

A component performance specification language for reusable specifications must be defined from the viewpoint of the component developer. During specification of a component, the component developer has no information about components connected to its required interfaces, its deployment platform, or parameter values passed to its provided services by clients. Because of this, the component developer has to provide a *parameterised* specification, which makes the influences by these external factors explicit.

Because a component can provide multiple services, which clients can possibly use independently, a component performance specification consists of a set of service performance specifications. For each provided service of a component, the component developer must provide a service performance specification. To be accurate, it has to include the following features:

- **Time Consumption:** As the execution times of individual component services in a software architecture add up to the overall execution time perceived by users, each component service has to specify its contribution to the overall execution time. However, component developers cannot directly specify service execution times using measurements, because such measurements would depend on their own deployment platform. On a different deployment platform, the provided services might exhibit different execution times, therefore such a specification would be inaccurate.

  As component developers do not know during specification on which deployment platform their components will be deployed, they have to specify the time consumption as a *resource demand*. For example, component developers can specify the processor resource demand as the number of CPU cycles needed. The number of CPU cycles does not refer to a timing value, however

it can only be converted into a timing value once the processing rate in terms of CPU cycles per second is known.

- **Memory Consumption:** Besides processing time, component services consume memory during execution. To derive memory-related performance metrics from a component-based performance model it is necessary that each component service specifies its memory consumption. For deriving execution time-related performance metrics, memory consumption is often negligible.

- **Access of Active Resources:** A component service can access different active resources, such as a processor or a storage device, during execution. Each of these resource can become the bottleneck of the system. To find such bottlenecks, it is necessary that each component service specifies resource demands for each accessed active resource.

- **Access of Passive Resources:** Besides active resources, a component service might also acquire and release passive resources, such as semaphores, threads from a pool, buffers, etc., during execution, which might lead to waiting delays due to contention with other concurrently executed services. Thus, a service performance specification should make such accesses explicit, so that the resulting contention effects can be investigated through analysis or simulation.

- **Calls to Required Services:** As described in the previous section, the execution time for executing required services adds up to the overall execution time of a service. The component developer does not know the execution time of required services, because it is unknown which components will provide these services if the component is used by third parties. However, the component developer must make calls to required services explicit in the service performance specification, so that their contribution to the overall execution time can be taken into account.

- **Control Flow:** The order of accessing resources or calling required service by a component service might change the resource contention in a component-based system. Therefore, for an accurate performance specification, a service performance specification should include the control flow between resource accesses and calls to required services in terms of sequences, alternatives, loops, and forks.

- **Parameter Dependencies:** As described in the previous section, the values of service parameters can change the time or memory consumption of a service, its accesses to active or passive resources, the amount of calls to required services as the control flow. Because the actual parameter values used by clients are unknown during component specification, component developers need to specify properties such as time or memory consumption in dependency to service parameter values.

- **Internal State:** If a component holds an internal state (in terms of the values of global, i.e., component-wide variables) during execution this state can influence the properties described above just like service parameters. Therefore, a component performance specification should include a notion of internal state, if it influences performance properties significantly.

Besides these performance-related properties, service performance specifications need to be composable isomorphically to the components in the software architecture and the specifications by different vendors need to be compatible to build an architectural model. Additionally, service performance specifications potentially have to abstract from the actual performance behaviour of a component, to remain analysable by performance solvers. The component developer creating a service performance specification must make a trade-off between the accuracy of the specification and its analysability.

Service performance specifications should not refer to network devices. Network communication in component-based systems should only happen between components when calling required services, but not inside components. Otherwise, a software component would not be a unit of deployment, if it would require multiple servers connected by network devices.

2.5 Related Work

This section surveys related work in the area of component-based performance prediction and usage modelling. Several other approaches have created reusable performance models for software components, but none of them has become common software industry practice. Chapter 2.5.1 introduces six performance prediction approaches, which tackle the problem of component-based performance modelling. The emphasis in this phase is on usage modelling and parameter dependencies in

these approaches. A comparison of these approaches to the newly introduced modelling language in this thesis follows after its description in Chapter 4.3.3.

Many other performance prediction methods besides these six approaches have been proposed, which analyse the performance of component-based systems. However, they are only loosely related to this thesis, as they use different component notions, do not aim a reusable performance specifications, or simply use conventional, monolithic performance models for the prediction. Chapter 2.5.2 briefly surveys these approaches.

This thesis also introduces a new modelling language to describe user behaviour (opposed to software behaviour). There are hardly any related approaches into this direction in the field of performance engineering. However, so-called "operational profiles" and "Markov usage models" from the area of software testing and reliability prediction are similar to the new modelling language. Therefore, Chapter 2.5.3 provides an overview of other models for user behaviour proposed in software engineering.

Additionally, this thesis contributes a static code analysis approach to generate performance models from code, and two transformations from newly introduced models to the performance domain. The related work for these two areas is discussed in Chapter 5.2 and 6 respectively.

2.5.1 Component-Based Performance Prediction Approaches

The following approaches propose performance prediction methods for component-based software architectures and create reusable component performance specifications (Table 2.1). The last column in this table refers to the features of the Palladio Component Model, which will be described in detail in Chapter 3 and 4. The following describes the related approaches in detail and tries to classify their component performance specification according to the requirements stated in Chapter 2.4.3. After the description of each approach, Table 2.2 summarises these requirements for each approach for easy comparison.

CB-APPEAR Eskenazi and Fioukov have introduced the APPEAR method in their dissertation [EFH04, EF04]. The goal of this method is to create a prediction model for arbitrary software entities (not necessarily software components) using simulation, measurement, and statistical methods. The resulting model can be used by software architects to predict the performance of systems, where the corresponding

Name	CB-APPEAR	CBML	CB-SPE	Hamlet	RESOLVE-P	ROBOCOP	PALLADIO
Literature	[EF04, EFH04]	[Wu03, WMW03, WW04]	[BM04a, BM04b]	[HMW01, HMW04]	[SKK01]	[BdWCM05, BCdW06a, BCdW06b, BCdK07]	[BKR08]
Domain	Distributed Systems	Distributed Systems	Distributed Systems	General	General	Embedded Systems	Distributed Systems
Design Model	Annotated Control Flow Graph	CBML/LQN	UML Sequence and Deployment Diagrams + SPT	-	-	ROBOCOP	PCM
Performance Model	Execution Graph + QN	LQN	Execution Graph + QN	Execution Graph	-	Task Tree	QN, Capra, SRE, LQN
Model Transformation	-	-	C-Code with XMI Input/Output	-	-	RTIE (ad-hoc)	Java (ad-hoc)
Performance Model Solver	Analytical, Simulative	Analytical, Simulative	Analytical	Analytical	-	Simulative	Analytical, Simulative
Prediction Feedback into Design Model	-	✓	-	-	-	-	(✓)
Automated Exploration of Design Alternatives	-	-	-	-	-	(✓)	-
Case Study	Toy Example (Car Navigation System)	Toy Example (Generic 3-Tier Arch.)	Toy Example (Software Retrieval System)	Toy Example (Unspecified Java Methods)	Toy Example (Stack)	Industry System (MPEG4 Decoder, etc.)	Toy Example (3-Tier Arch.: MediaStore)
Implementation Support	-	-	(Code Generation of UML tool)	-	-	-	Java/EJB Code Generation
Tool Support	-	Jlqndef, LQNS, LQSiM	CB-SPE Tool Suite (incl. ArgoUML, RAQS)	-	-	RTIE	PCM-Bench
Current Status	Completed in 2004	Completed in 2004	Abandoned in 2005	Active (2007)	Abandoned in 2004	Active (2007)	Active (2007)

Table 2.1: Component-Based Performance Prediction Approaches

software entity is reused in a slightly adapted way. In [EF04], they describe which changes to the software entity are allowed to keep the prediction model valid.

The APPEAR method has also been extended to be applied on component-based systems [EFH04], and will be referred as CB-APPEAR in the following. The problem scenario described by the authors is slightly different from the other component-based performance prediction approaches. The approach requires an implementation of all components within an architecture to enable measurements. Software architects or performance analysts carry out these measurements and build performance prediction models for each component service. The resulting models could in principle be reused in other architectures, which use the same components or slightly adapted versions of them. However, this is not intended nor described by the authors, as they focus on making predictions for the systems they have built the models from.

The notion of a software component is similar to Szyperski's definition, as CB-APPEAR components have provided and required interfaces with multiple services [EF04, p.150]. The approach does not consider composite components, but only assemblies of components bounded by connectors (i.e., the component model is not hierarchical). There is no formally defined meta-model for the component

specification and also no tool support to create such specifications. In their case studies, the authors use the COVERS simulation framework [BKR95] to build simulation models.

The CB-APPEAR process consists of three steps: (i) creating a prediction model for individual component operations (i.e., a set of instructions executed by a component), (ii) creating an annotated control flow graph for single component services, and (iii) creating and simulating a full application model, which may include executing individual component services concurrently.

For creating the prediction models for single component operations in the *first* step, a performance analyst identifies performance-relevant parameters (not necessarily from the component interfaces). Then the component is executed multiple times (without concurrency) using a large set of use cases. With execution time measurements from these test runs, the performance analysts builds a formula based on statistical regression methods. The formula includes the formerly identified performance-relevant parameters as variables. This parameterised time consumption specification does not refer to different resources, such as CPU or hard disk and also does not include the use of passive resources.

In the *second* step, the performance analyst builds an annotated control flow graph for each component service, whose nodes model either component operations with the time consumption formula from the former step, or calls to other component services. Supported control flow constructs are sequence, alternative, and loops. Forks that model concurrent control flow inside a component service are not allowed. The approach also explains determining parameter dependencies for branch probabilities and loop iteration numbers. However, the parameters in these dependencies need not refer to service parameters, but refer to the performance-relevant parameters determined in the former step.

Once the performance analyst has modelled all control flow graphs they are connected to each other in a so-called activity composition in the *third* step. The performance analysts also defines the execution frequency of the services (i.e., an open workload). The resulting model can be used as input for analysis or simulation solvers depending its underlying assumptions. There is no standard procedure for solving the models, as the authors leave this to the choice of the performance analyst. In a case study [BMdW+04], the authors determine the schedulability and worst-case execution times of a component-based car navigation system.

The approach supports parameter dependencies for branch probabilities, loop iteration numbers and resource demands. However, these dependencies are de-

termined ad-hoc and their validity in other scenarios is unclear. It is also unclear, whether the resulting service performance specifications can be reused. Memory consumption is considered by the approach verbally, but not explicitly demonstrated. There is no account for internal state of component or passive resources.

CBML The Component-Based Modelling Language (CBML) is an extension to LQNs to make parts of LQN models replaceable. It has first been introduced by McMullan [McM01] as a BNF-grammar and later been implemented as an XML schema by Wu et al. [WMW03, WW04]. Wu's Master thesis [Wu03] describes CBML in detail. LQNs have already been mentioned in Chapter 2.3.2 and will be described in detail in Chapter 6.4.

LQNs include a concept called "task", which expresses the behaviour and resource demands of a software entity (details in Chapter 6.4). CBML uses tasks as software components and adds a so-called *slot* around them. A slot may contain a number of interfaces representing the provided or required services of a software component (called "in-port" and "out-port"). CBML therefore uses the component concept of UML2, where components consist of interfaces associated with a port. Slots make the CBML components replaceable with other CBML components conforming to the same slots. Using slots, CBML components can also be nested to express composite components.

Besides a "task", a CBML component may also contain a set of processors, where the task gets executed. These processors inside components are rather placeholders, which have to be bound to actual processors in the LQN surrounding the component once the component is plugged into a slot. Therefore, a slot contains different types of *bindings*. It can bind ports to each other to connect a CBML component to an architecture. It can bind processors to each other to specify that a component uses a specific processor. And finally, a binding may contain a set of parameters, where values are bound to parameter names.

Parameters in CBML components usually do not refer to the input parameters of component services (which cannot be expressed in CBML), but rather to arbitrary performance attributes of a CBML component. For example, they can be used to adjust the thread pool size for a component or to adjust its resource demands. Thus, they reveal additional internals of a component besides the service interface. The values of these parameters are of type String, therefore it is difficult to make a type-conform assignment if different performance attributes (e.g., real number for resource demands, integer for thread pool size) inside a component are parame-

terised. The parameterisation of control flow properties, such as branch probabilities or loop iteration number is not mentioned in this approach.

Besides slots, bindings, and parameters, CBML inherits all other modelling concepts from LQNs. Time consumptions in LQNs are resource demands to processors, their amount is specified by mean values of exponential distributions. Memory consumptions are not supported by LQNs. LQNs can model active as well as passive resources, and may include control flow concepts such as sequence, alternative, loop, and fork. LQNs are particularly useful to model asynchronous communication in distributed services, as they include special concepts for this. LQNs do not consider any data values or internal state of software entities.

The CBML language builds on the resource function capture approach by Woodside et al. [WVCB01], which aims at determining parameterised resource demands. It uses a testbed to repeatedly execute software components with different parameters and measures their execution times and calls to required services. Using statistical regression techniques on the measurement results yields functions for resource demands. These functions could be used in combination with CBML, which is however not shown in the respective case studies [WW04].

CB-SPE The Component-Based Software Performance Engineering (CB-SPE) approach by Bertolino and Mirandola [BM04a, BM04b] uses UML extended with the SPT profile [Obj05a] as design model and queueing networks as analysis model. The CB-SPE framework tries to adhere to standards such as the UML component definition and the SPT profile where possible and uses freely available modelling tools (ArgoUML) and performance solvers (RAQS). The approach is divided mainly into two layers: the component layer and the application layer.

The *component layer* lets component developers specify the performance of their components and put these specifications into publicly accessible repositories for software architects. The performance specification is a function $Perf_{C_i}(S_j[envpar]^*)$ for a component C_i and a list of services S_j. $Perf$ is a performance index, such as an execution time or communication delay. With a list of environment parameters ($[envpar]$), which for example can refer to network bandwidth, CPU speed, etc., the function can be adjusted by software architects for different platforms. The list of services S_j includes provided *and* required services. Therefore, component developers can also specify the resource demand requested from required services. The parameterisation does not involve service input or output parameter, but only refers to properties of resources. It is not possible to specify what required services specific

provided services will call. In the *application layer*, the software architect performs three steps: component pre-selection, modelling/annotation, and analysing the results of the performance solvers. For pre-selection among functionally equivalent components, the software architect instantiates the $Perf$ function retrieved from a repository with the current platform parameters and weights the resulting performance indices according to specific performance goals (e.g., max. throughput, min. response time, etc.). For example, if the software architect is more interested in analysing response times, the corresponding performance index may be weighted higher than other performance indices. This enables selection of a matching component.

In the second step, the software architect models the control flow through the planned component-based software architecture using UML sequence diagrams. This might be hard to realise using black-box components, because the visible component interfaces and the provided performance indices by the component developer are insufficient to determine which required services will be called. Besides the control flow, the software architect annotates the sequence diagram with the performance indices from the component developer. For example, the execution time of a provided service is included into the model using a PAStep annotation from the UML SPT profile and using the performance index calculated from the function provided by the component developer. Additionally, the software architect creates deployment diagrams to model the resource environment and annotates the included resources for example with scheduling disciplines according to the SPT profile.

The complete model, which consists of sequence and deployment diagrams, is then input for the CB-SPE tool, which generates SPE execution graphs (EG, cf. Chapter 2.3.3) or queueing networks (QN, cf. Chapter 2.3.2) from them. Using an own EG solver, performance metrics for single-user scenarios can be determined. Using the RAQS QN solver, performance metrics for multiple-user scenarios can be determined from the model.

The performance model mainly inherits from the UML SPT profile. The profile supports arbitrary distribution functions for specifying time consumptions, but the QN solver can only handle mean values of exponential distributions. CB-SPE has no support for determining memory consumption. The control flow modelling is realised with sequence diagrams, thereby supporting sequences, alternative, loops, forks, and also asynchronous communication.

A problematic feature in CB-SPE is the fact that the software architect needs to specify the control flow through the architecture instead of the component devel-

opers specifying the control flow of their individual components. If the software architect replaces one component by another, the whole architecture has to be specified again. In addition, there is no possibility to specify concurrency inside components. The parametric dependencies supported by CB-SPE remain rather fuzzy, as the authors refer to other approaches to determine appropriate performance indices for software components. There is no support for modelling internal state, and a component can only use a single resource.

Besides the modelling deficits, it remains unclear, which elements of the UML SPT profile can be used by the software architect, as the QN solver does not support the whole expressiveness of the profile (e.g., general distribution functions for timing values or some of the scheduling disciplines). Nowadays, the CB-SPE tool suite based on ArgoUML is rather outdated and no longer supported by the authors.

Hamlet The approach by Hamlet et al. is no classical performance modelling approach, but originates from the area of software testing. It was first proposed for reliability prediction of component-based systems [HMW01], and later for performance prediction [HMW04]. The authors try to create a fundamental theory of software composition, which enables reasoning on different properties such as correctness, reliability, performance, security etc. based on specification of individual components.

Although the authors reference Szyperski's component definition, they use their own very restricted component definition to reduce the complexity of the theory. A software component in this approach is a mathematical function with a single integer parameter. If a component is composed to another component, it sends its output (the result of computing the function) to the other component, which uses it as input. Therefore, component composition always follows a pipe-and-filter pattern in this method.

The method of performance prediction for systems composed of such components consists of several steps. First, after implementing a component, each component developer specifies a set of subdomains for it. A subdomain is a subset of the component's input domain. The component developer should specify each subdomain in a way that calling the component with any parameter from this subdomain yields a similar execution time. For example, in the most trivial case, the component developer simply divides a component's input domain into two subdomains, one for slow execution time, one for fast execution times.

Second, the component developer measures the execution time for invoking the

component with parameters from each subdomain and stores the measurement result for each subdomain in a repository. From this repository, software architects can retrieve these measurements and also the components. The authors do not consider that component developer may use a different deployment platform than the software architect, therefore the measurements by the component developers may be invalid for the software architect's deployment platform. They also do not spread execution time onto different resources (e.g., CPU, hard disk) and ignore passive resources.

In a third step, the software architect specifies an operational profile consisting of a probability for each subdomain of the first (pipe-and-filter pattern assumed) component in the planned architecture. Then, the software architect retrieves the desired components for the architecture from the repository and executes the first component using the specified operational profile. For each subdomain of the first component, this results in a probability of calling a particular subdomain of the second component. Using these probabilities, the second component can be execute to determine how it propagates the requests.

If the probabilities for calling each subdomain in the architecture have been measured, the execution time of the overall architecture can be determined by adding the measured execution times from the component developers weighted by the call probabilities. Besides sequential composition, Hamlet et al. also show in [HMW04] how this approach can be applied for components called conditionally (alternative) or iteratively (loop). The whole approach is not restricted to execution times of components, but can be applied to reliability, security, safety, correctness, and other properties.

The approach only specifies time consumption as constants and does not consider distribution functions. It does not consider memory consumption. There are no explicit parameter dependencies in this approach, and components cannot execute concurrently. Branch probabilities or loop iteration number have to be determined via measurements for each prediction.

The most critical aspect of this approach is the execution of the components by the software architect to determine the propagation of inputs. This requires the software architect to deploy the components of the whole component-based system in advance to conduct the prediction. This is not desirable, because it reduces the approach to performance testing and invalidates the value of the created models.

As the authors point out, it is crucial to determine "good" subdomains for this approach to yield accurate predictions. Techniques from black-box testing are sug-

gested for this, but it is usually not possible to test the component for all possible input value to gain an accurate subdomain partitioning.

Hamlet is one of the few authors, who investigates stateful software components and the impact of the internal state on performance properties. The approach suggests to model internal state as additional inputs to a component's input domain. However, this might easily lead to a combinatorial explosion of the number of required test cases. Therefore, it is recommended to use system-wide internal state only if absolutely necessary, and treat stateful components with formal analysis instead of testing.

RESOLVE-P Sitaraman et al. [SKK+01] have proposed a dialect of the RESOLVE specification and implementation language [EHL+94] for software components to additionally express performance properties. The following will refer to this approach as "RESOLVE-P" (P for performance). The authors tackle the challenge of component performance specification from the perspective of computational complexity theory.

Their aim is to provide specifications of the time and memory consumption of component services in a refined big O-notation. They want to make assertions about the asymptotic form of time and memory consumption and later formally verify those assertions. It is not intended to derive for example accurate response times or resource utilisations as in the other approaches described before.

The component notion of RESOLVE-P is rather related to modules and objects. The authors use a generic stack object to illustrate their approach. There are no explicit required interfaces or calls to other components in their examples. RESOLVE specifies the functionality of a component with a list of service signatures and a pre- and post-condition for each service.

As problem motivation, the author point out that classical big-O notations are not helpful for generic components with polymorphic data types. For example, specifying the time consumption to copy a generic stack S with $O(|S|)$ (linear complexity), where $|S|$ is the current size of the stack, is inaccurate, because the elements within the stack might be complex objects such as trees, which could require logarithmic or polynomial complexity for copying. Therefore, Sitamaran et al. propose to first extend classical big-O notations, which are defined on the domain of natural numbers, to arbitrary mathematical spaces. However, in their examples they still use natural numbers for their big-O notations.

The specification of time and memory consumption of a component is bound to

a particular component implementation. In [SKK+01], each provided service of the example stack specifies its time consumption and memory consumption.

The specified time consumptions for the stack operations are functions involving the time consumptions for initialising and destroying the inner elements of the stack. They are therefore more refined than a classical big-O notation, which would not take the structure of inner elements into account. It is not defined whether the time consumptions can be specified as constants or distribution functions. The time consumption does not refer to different active resources. The approach also does not account for passive resources. Because of the missing calls to required services and the single assumed resource, there is also no control flow reflected in these specifications.

For memory consumption, the authors propose an adapted big-O notation, which includes a fixed coefficient. For example, a memory specification in $O(2n)$ would be in a different class than $O(3n)$ in the proposed notation, while in classical big-O notation both specification would fall into the linear complexity class. The authors motivate this adapted big-O notation by claiming that the coefficient often has a significant impact on the memory consumption, which would be inaccurately expressed by the classical notation.

The approach still remains abstract and only serves to explain some problems with classical big-O notations on a generic stack. Without the account for different resources and the missing required interfaces, the component notion is rather limited. The example performance specification of the stack refers to the internal state of the component (the stack size). Therefore, this is one of the few approaches that includes the influence of the internal state on the performance of the component albeit in a limited fashion.

ROBOCOP Bondarev et al. [BMdW+04, BdWCM05, BCdW06a, BCdW06b] have developed a performance prediction approach for the ROBOCOP component model [Gel] aiming at analysing embedded systems.

The component definition of ROBOCOP is mostly in line with Szyperski's definition, however it uses different terms than established (for example, a "component" is called "service"). This description uses the established terms, however. Components have provided and required interface including multiple service signatures. Composite components are not supported. Besides a functional specification, ROBOCOP also provides a resource specification and a behaviour specification for each component and couples an executable implementation with the specification.

The process of performance prediction is carried out by component developers and software architects. Component developers provide parameterised specifications of their component and put them into a repository. Software architects retrieve these models and themselves model an application scenario consisting of a component assembly and a number of performance-critical usage scenarios. They also instantiate the parameters specified by the component developers.

For the prediction, the tool Real-Time Integration Environment (RTIE) compiles the application scenario, component resource models, and component behaviour models and converts them into a so-called task tree. RTIE then simulates the execution of a specific scenario and creates a task execution timeline as output, which illustrates the response time, blocking time, number of missed deadlines of each task, and utilisation of each resource.

All ROBOCOP model elements use constant values instead of random variables to specify for example resource demands or component behaviour. There are no stochastic annotations, as the model always reflects a single request through the architecture. Therefore, the control flow specification for example does not contain branching points with probabilistic branches, as a deterministic program always chooses a single branch.

The component developer can specify the time consumption of a component service as a fixed time value for a processor or a network. There is no support for platform-independent resource demands. The component developer can also specify the memory consumption of a component service by defining the amount of memory claimed and the duration of the claiming. There is always a single CPU time, a single network time, and a single memory consumption for each service, and it is for example not possible to specify a CPU access after a memory access. Access to passive resources besides memory is supported by specifying critical regions.

Calls to required services may be executed synchronously or asynchronously. ROBOCOP also supports specifying input parameters for calls to required services. As control flow constructs, only sequences and loops (with constant iteration numbers) are supported. There are neither branches nor forks, as the service behaviour is always only a single run through the architecture and ROBOCOP does not allow concurrency inside components.

The approach is particularly strong on specifying parameter dependencies (described in [BdWCM05, BCdW06b]). In this area, it is most related to the parameter dependency model proposed in this thesis. Component developers can specify resource demands, loop iteration numbers, and input parameters to required services

in dependency to a service's input parameter values. When software architects later provide values for the parameters, RTIE can solve the parameter dependencies.

ROBOCOP allows only integers to specify parameter values. There is no support for other data type values or random variables. However, it is possible to specify a minimum and a maximum parameter value to define a range. In [BCdW06b], it is also possible to specify the byte size of a parameter or the number of elements if it models an array. The model also supports specifying component-wide parameter values, which can be used by all behavioural specifications of its services. They can be used to express a configuration of a component and can be seen as a static internal state model.

ROBOCOP only supports dependencies to input parameter values and neglects output parameter values, which could be the result of calling required services. Therefore the possible specification of component interaction is rather limited, as components can exchange data only into a single direction. In [BCdW06b], the model has been extended to also allow the specification of return values for each call to a required service. However, the concrete return value is usually unknown to the component developer specifying the component. It can also not be supplemented automatically by tools by taking the values from other components once the component is placed in an architecture, because the service specification lacks a facility to set return values.

Conclusion Concluding, Table 2.2 again compares the service performance specifications of the prediction approaches described before.

None of the approaches supports all the requirements stated in the former section. All approaches include some form of parameterisation, but only RESOLVE-P and ROBOCOP explicitly support the specification of dependencies to parameters declared in component interfaces. Other approaches, such as CBML or CB-SPE do allow a variation of certain performance attributes (such as thread pool sizes or resource processing rates) and consider this as parameterisation. As they do not include component interfaces specifying signatures with input and output parameters, they do not support the specification of dependencies to these parameters.

2.5.2 Other Component-Based Performance Analysis Approaches

There are many other approaches, which analyse, model, measure, or predict the performance of component-based software systems (survey in [BR06]) besides ap-

Name	CB-APPEAR	CBML	CB-SPE	Hamlet	RESOLVE-P	ROBOCOP	PALLADIO
Literature	[EF04, EFH04]	[Wu03, WMW03, WW04]	[BM04a, BM04b]	[HMW01, HMW04]	[SKK01]	[BdWCM05, BCdW06a, BCdW06b, BCdK07]	[BKR08]
Time Consumption	Platform-Dep. Overall Run Time (Constant)	Platform-Indep. Resource Demands (Exp. Dist, Mean Value)	Platform-Indep. Resource Demands (Exp. Dist, Mean Value)	Platform-Dep. Overall Run Time (Constant)	Extended O-Notation	Platform-Dep. Overall Run Time (Constant)	Platform-Indep. Resource Demands (Gen. Dist., pdf)
Memory Consumption	✓	-	-	-	Extended O-Notation	Constant	-
Access of Active Resources	-	✓	✓	-	-	✓ (only CPU, network)	✓
Access of Passive Resources	- (only memory)	✓	- (only spec., no analysis)	-	-	✓	✓
Calls to Required Services	?	Asynch/Synch	Asynch/Synch	Synch	-	Asynch/Synch	Synch
Control Flow	Sequence, Alternative, Loop	Sequence, Alternative, Loop, Fork	- (Control Flow specified by SA)	- (Control Flow specified by SA)	-	Sequence, Loop	Sequence, Alternative, Loop, Fork
Parametric Dependencies (see Section 4.3.4 for more detail)	(✓) (for branchs and loops, yet purpose unknown)	Not for service parameters	Not for service parameters	- (Input Propagation via Measurements)	(✓) (limited to time/memory consumption)	(✓) (limited to constant input parameters)	✓
Internal State	-	-	-	- (Treatment as additional Input)	(✓) (limited to time/memory consumption)	- (Constant Configuration Parameters)	- (Component Parameter)

Table 2.2: Comparison of Service Performance Specifications

proaches for monolithic systems (survey in [BDIS04]). However, these component-based approaches are only loosely related to this thesis, as they use different definitions of software components, simply provide special measurement facilities without modelling, or rely on different underlying assumptions. Many of these approaches do not aim at reusable performance models for components, but instead just analyse the performance of component-based systems with conventional, monolithic modelling methods without exploiting the component paradigm.

For completeness, the following briefly surveys these approaches to delimit them from the formerly described approaches and the work presented in this thesis. This should help the reader not to confuse these approaches with the more specific related work for the same problem as in this thesis, which has been presented before. The following approaches can be broadly categorised into model-based methods (M1-M6) and measurement-based methods (M7-M15) and are described in chronological order of their appearance in the following. Many of these methods are also described in the survey by Becker et al. [BR06].

Model-based Methods **M1.** Gomaa and Menasce [GM01] create performance models for component-based systems using UML class diagrams annotated with a proprietary XML-based notation. They put special emphasis on modelling component interaction patterns, such as client/server, synch/asynch connectors, as a

well as single/multi-threaded servers. However, they build a single model for a whole system instead of individual, replaceable parts for single components. They explicitly model probabilities for calling required services after invoking provided services, and consider the size of network messages for the performance prediction. There is no parameterisation of the modelled values, which renders them useless for reusability. They map the resulting model to a queueing network and solve it analytically.

M2. Hissam et al. [HMSW02, HMSW03] present a method called Prediction Enabled Component Technology (PECT), which is based on the COMTEK component technology. They determine certifiable component performance specification using measurements on a stable resource environment and applying statistical methods. The author demonstrate the applicability of their approach with a soft real-time latency prediction for a CD player. This approach remains rather abstract and generic, as it uses simple performance specifications and does not provide a detailed description of the case study.

M3. Zschaler et al. [Zsc04, RZ07, Zsc07] focus on the specification of performance properties of software components, but do not carry out performance predictions. The authors are more interested in developing a formal model for performance specifications, which can be checked statically on component assemblies. The authors use different notations including CQML+ [Aag01] and Z specifications, which however do not include control flow or parameter dependencies.

M4. The UML SPT profile [Obj05b] and its successor the UML MARTE profile [Obj07a] allow the annotation of arbitrary UML elements with performance properties. Combined with the UML2 component model, developers could use these profiles to create performance specification of software components. However, as pointed out in [KHB06], the UML and both profiles provide no facilities to model the values of service parameters and specify parameter dependencies needed for reusable performance models based on them. In general, the UML targets object-oriented design and monolithic architectures and provides limited support for replaceable, reusable models of software components [BKR08].

M5. DiMarco et al. [DI04, DiM05] use UML2 component diagrams and sequence diagrams annotated according to the UML SPT profile to model the performance of a component-based system. Software architects may compose the specifications by component developers. The model allows a parameterisation for the number of users inside a system, but is bound to the limitations of UML SPT as described above (i.e. there is no notion of dependencies to service parameters). The authors

generate a multi-chain queueing network from the model, which they solve using numerical techniques.

M5. Grassi et al. [GMS05, GMS07a, GMS07b] have developed the Kernel Language for Performance and Reliability Analysis (KLAPER), which explicitly targets component-based systems. However, the goal of this language is not to create reusable models for individual software components, but to provide an intermediate language to ease the implementation of model transformations. Chapter 2.3.4 further describes this language.

M6. Kounev [Kou06] presents a capacity planning study of a component-based software system (SPECjAppServer2004). Kounev measures the performance properties of an implementation of the system, and builds a performance model based on them. The model is a Queueing Petri-Net (QPN, also see Appendix B.2). However, although Kounev models and measures a component-based system, the component structure is not reflected in the resulting performance model. Components in this approach are application servers and databases. The method is not specific for component-based systems, just applied on such a system as a case study.

Measurement-based Methods **M7.** Cecchet et al. [CMZ02] conduct an intensive measurement study of an EJB-based system called RUBIS. They use different application server and point out factors, such as container- and bean managed persistence, which significantly influence the performance and scalability. They especially focus on application servers and their configuration parameters and find that the contribution of individual EJBs to the overall execution time is marginal. Although there is no building of a performance model in this study, it provides interesting hints on important factors, which should be included in an accurate prediction model for J2EE applications.

M8. Yacoub [Yac02] also conducts a measurement study of a component-based system. The approach is more general and not tied to a particular middleware platform, such as J2EE or .NET. It explicitly considers black-box components and the needed monitoring proxies for their interfaces to make measurements. The author also discusses the automation of the approach, but does not provide a formalisation of the concepts or tool support.

M9. Denaro et al. [DPE04] target early design time performance predictions for J2EE applications, but use prototyping instead of modelling. They argue that individual components have little impact on the overall performance and that the middleware causes the longest execution times. Based on this assumption, they

generate component stubs from a given software design, which can be deployed different application servers. The resulting prototype application can be monitored for its performance properties, which the author claim is sufficient to assess the real applications performance properties during an early design phase.

M10. Diaconescu et al. [DMM04] propose a measurement approach for J2EE/EJB systems, which they have integrated into their COMPAS framework. It uses a proxy layer to instrument EJBs and set up monitoring facilities. The overall goal of this approach to detect performance problems in a component-based system during runtime and then adapt the system to ensure certain service level agreements. Therefore the system can be considered as self-healing. The approach uses replication of components to adapt the systems on high load levels. There is no model building is this approach.

M11. Chen et al. [CGLL02, CLGL05] investigate the performance influencing factors of CORBA, COM+, and J2EE systems. As an example, they analyse a J2EE server with a bottom-up testing approach. The method requires developers to create a minimal application consisting of a simple EJB with a read and a write service and a database system, which includes a single database table. Using this prototypical spike, they measure reading or writing to the database through the EJB. With the measurement results, they build a simple formula, which parameterises the response time of the application over the number of available server threads and the number of concurrent requests. This formula allows determining the optimal setting for the size of the application server's thread pool.

M12. Liu et al. [LFG05] also analyse the performance of J2EE systems during early development stages. Their approach uses simple performance models, which are built based on measurements. The approach divides the modelling effort into two parts: (i) benchmarking an application server on a given hardware platform to create an application independent performance profile, and (ii) creating a description of the application and completing it with performance annotations obtained from additional measurements. The authors do no create reusable models for individual components, but a single model for the whole application. The resulting application model is a simple queueing network including a request queue, a container queue and a data source queue, where the service demands are determined from the description of the application. The model allows determining the best persistency strategy and assessing the scalability of the application.

2.5.3 Usage Modelling

Besides a component performance specification language with parameter dependencies, this thesis also proposes a new language (called PCM usage model) to model user behaviour and the values of input parameters (Chapter 4.2). It is aligned with the component performance specification language, as both are part of the PCM.

There are hardly any comparable usage models in the area of performance engineering. Many performance prediction methods use formal models, such as queueing networks, which model user behaviour only implicitly. They contain user arrival rates or the user population and the service demand for each service center in the queueing network thereby mixing user behaviour with system behaviour. However, some of the recent performance prediction methods model user behaviour explicitly with annotated UML diagrams. One of the following paragraphs briefly describes and evaluates such usage models based on the UML.

Approaches from other areas than performance engineering, such as reliability engineering and software testing, model user behaviour explicitly. The PCM usage model has adopted concepts from these approaches, therefore they qualify as related work. Thus, the following will describe Operational Profiles and Markov Usage Models used in these areas. This overview has also been published in an extended version in [Koz05].

Operational Profile John Musa has advocated the use of operational profiles to guide software testing [Mus93]. A formal definition has been given by Hamlet (Definition 12).

Definition 12 Operational Profile [HMW04]

Let a software system have n major functions that exhaust the input space D, thus it has the functional subdomains $S_1, S_2, ..., S_n$, where $D = \bigcup_{i=1}^{n} S_i$. An **Operational Profile** is a vector $< p_1, p_2, ..., p_n >$ with $\Sigma_{i=1}^{n} p_i = 1$ and gives probabilities p_i that input will fall into S_i, i.e., the probability is p_i that the ith function will be used.

In this case, a software system is viewed as a black-box providing several services (synonymous to functions) to the user. The operational profile simply assigns a probability to each of these services based on their anticipated or already measured usage. With an operational profile, testing can focus on the services with the highest probabilities, as they contribute the most to the overall system reliability.

Musa [Mus93] described the practice of using operational profiles at AT&T and sketched a five-step process to define such a profile for larger systems. In each step, the profile is successively refined and the result of step five is the operational profile. The process begins with specifying a *customer profile* for the user groups or institutions that will use the system. In the second step, each customer probability is broken down into a *user profile*, which assigns probabilities for the different types of users of the customer (e.g., admins, regular users). In the third step, each user probability is broken down in to a *system-mode profile*, which assigns probabilities according to different system modes (e.g., overload mode, normal operation). Each system mode can again have several functions, therefore the forth step provides a *functional profile*.

Up to this point, the profiles do not refer to implementation artefacts, but only design artefacts. Functions can be implemented by (possibly multiple) operations, thus the final step refines the functional profile to an operational profile. The resulting operational profile, which typically can include several hundred operations, is then input for test case selection (also see [AW95]). The occurrence probabilities of the profiles can be determined from experience with similar systems, via monitoring if the system is already running, or simply via estimations, which however might be inaccurate.

Woit [Woi94] deems operational profiles insufficient if a system holds a state, and the occurrence probabilities depend on this state and the call history. She specifically targets testing software modules, which encapsulate a state. Also, Markov usage models are not sufficient in this case, because they capture only the dependency to the current state but not to the call history. Woit therefore defines a new model, which captures execution history of user calls, so that a new call probability can be determined given a particular call history. As this model recognises context-free languages, it is more expressive than operational profiles or Markov usage models. She also covers test case generation and reliability estimation from this model in her PhD thesis, but an industry case study is missing and the applicability for non-trivial systems is unknown.

Voas [Voa98, Voa99, Voa00] has argued that using operational profiles for software testing tends to neglect seldom used, but critical operations. An extreme example would be the emergency procedure in a nuclear power plant, which would hardly be tested based on its occurrence probability when using an operational profile. Furthermore, Whittaker and Voas [WV00] discuss that developer often falsely estimate actual user behaviour, because they do not correctly anticipate how users

will execute a system. Using operational profiles does not account for concurrently running processes or data present in the system, which might invalidate testing results and reliability estimations based on them.

Recently, Gittens [GLB04] has proposed an extended operational profile, which also includes a specification of the data used by the system in addition to the classical operational profile. It does however neglect call histories. Simmons has recently reported on some best practices in determining operational profiles [Sim06].

As already stated, operational profiles neglect call histories, current state, and also input parameters to functions. These factors may all have an influence on the performance or reliability of a software system, and therefore should be included in such a model.

Markov Usage Model To express sequences of service invocations by users, Whittaker et al. [WP93] have proposed modelling the usage of a system with discrete finite Markov chains. A formal definition has been given by Gutjahr et al. (Definition 13).

Like operational profiles, Markov usage models describe the behaviour of users and treat the software system as a black-box. The approaches using these models should not be confused with other approaches, which model the control between software components as Markov chains. Whittaker and Poore [Whi92, WP93, WT94] first used Markov usage models for test case generation and statistical software testing. They were later integrated into the cleanroom software engineering approach [MDL87, PTLP99].

For determining the transition probabilities of the Markov chain, which are also called "usage profile", the same methods as for operational profiles shall be applied. If a running system is available, current user behaviour can be monitored to measure the transition probabilities. If the system is still under development, the transition probabilities must be estimated based on experience. If no experience is available, a uniform distribution for transition probabilities has to be assumed.

Wohlin and Runeson [WR94] propose an extended, hierarchical Markov usage model, which allows reusing the Markov chains for using particular services, if they are used multiple times in a scenario. This allows reducing the state space of the Markov chain, thereby increasing tractability. Menasce et al. [MAFM99] used a Markov usage model in a performance modelling context and modelled the behaviour of website users. They evaluated logs of webservers to determine transition probabilities. Farina et al. [FFO02] show how to reduce the state space of Markov

Definition 13 Markov Usage Model [Gut00]

A **Markov Usage Model** is a Markov chain with a unique initial state, symbolizing program invocation, a unique final state, symbolizing program termination, and other states symbolizing intermediate usage or processing state of the program under consideration. The Markov chain can be represented by a directed graph $G = (V, A)$, and a function $p : V \times V \rightarrow [0, 1]$ with the following properties:

- $V = \{1, ..., n\}$ is a set of nodes, representing the program states (e.g., program invocation, program termination, input/output screens).
- A is a set of arcs, representing state transitions which always correspond to specific operations of the program. An arc from state i to state j is denoted by the ordered pair (i, j). Multiple arcs between i and j are not allowed.
- $p(i, j)$ is the transition probability from state i to state j, if (i, j) is an arc. Otherwise, we set $p(i, j) = 0$. The transition probabilities satisfy the conditions $0 \leqslant p(i, j) \leqslant 1$ and $\Sigma_{j=0}^{1} p(i, j) = 0$.

The values $p(i, j)$ can be represented in a comprehensive form by a matrix $(p(i, j))_{i,j}$ It is always supposed that state 1 is the initial state and state n is the final state. State n is assumed to be an absorbing state, i.e., it cannot be left anymore: $p(n, n) = 1$ and $p(n, j) = 0$ for $j \geqslant n$. Furthermore, it is assumed that each node $i \in V$ is reachable from node 1, i.e., there is a directed path in G from node 1 to node i.

usage models by using stochastic automata networks instead.

Doerner et al. [DG00] describe several problems when using Markov usage models. The number of loop iterations for repetitive user behaviour is always geometrically distributed in a Markov usage model, which is however not representative of reality. It is not possible to model consumer/producer situations, which occur frequently in reality. Furthermore, Markov usage model do not model parameter values supplied by users explicitly.

There are also several approaches that model the interactions between software components instead of the transition between user states with Markov chains [GPT01]. These methods consider the transition probabilities between components as the usage profile [RSP03]. They aim at reliability prediction for component-based software architectures. The methods proposed by Cheung [Che80] and Reussner [RSP03] are among them. They assume that component behaviour follows a plain Markov chain, which is however violated by many existing software components. Furthermore, they assume that the transition probabilities can be determined via measuring an implementation of an architecture, which contradicts the idea of model-driven predictions before implementation.

Usage Modelling in UML Software developers can use UML use case and activity diagrams to model user behaviour. With the UML SPT profile [Obj05b], they can annotate use cases with workloads and use activity diagrams to model user arrival rates and usage scenarios. The profile also allows annotating actions from activity diagrams with probabilities or waiting delays to capture user behaviour in a probabilistic way. Furthermore, UML2 activities include object nodes to model data flow and express transferring parameters between actions. Therefore the UML provides expressiveness to model usage for performance predictions.

However, specifying usage with UML has at least two disadvantages. First, the language is still limited in its expressiveness, as it is not possible to model the size or values of parameters, which can have a significant impact on performance. It is furthermore not possible to model loop iteration numbers with probability distributions, because the SPT profile allows only constants for the number of repetitions of an action. Marzolla [Mar04] describes some of the difficulties and limitations when modeling with the SPT profile. Even the upcoming UML MARTE profile [Obj07a] will not solve these issues.

The second disadvantage of using UML for specifying usage is its inherent complexity. The current UML2 specification spans over more than 1000 pages, the UML

SPT specification has more than 250 pages. Even if it is not necessary to comprehend the whole specification to model usage for performance predictions, a profound knowledge of UML activities and the SPT profile is necessary. Therefore, it is hardly possible for persons without a technical background to model user behaviour in the UML.

However, the usage of a system is mainly determined by non-technical factors, therefore domain experts have to provide the needed information. In current practice, software developers interview domain experts for the expected usage of a system and capture this information in models. It is desirable that domain experts can create usage models themselves with a restricted, domain-specific language without having to learn the intricates of UML, which includes many constructs outside their domain.

2.6 Summary

This chapter has presented the foundations of this thesis. As the thesis contributes in the area of component-based performance engineering, Chapter 2.1 explained the basics of CBSE. It defined and described both software components and software architecture and described a process model to developer component-based systems. Chapter 2.2 then described model-driven software development, because this thesis introduces new modelling languages. It defined concepts such as meta-model, domain specific language, and model transformation. Chapter 2.3 contained basics about performance engineering. It described common performance metrics and performance analysis methods. Furthermore, it included a brief overview of different kinds of performance models.

Chapter 2.4 then brought together the three formerly described areas and described the motivation and intricates of component-based performance engineering. In particular, it established a list of requirements for a performance specification language for software components. Finally, Chapter 2.5 analysed existing approaches for component-based performance modelling according to the requirements. It pointed out that the existing method provide limited support for specifying usage profile dependencies.

Chapter 3

Basics of the Palladio Component Model

This thesis proposes new models for user and component behaviour reflecting the performance influence of input and output parameters as part of a usage profile. These models are embedded into the Palladio Component Model (PCM), which is well suited to express these influences as it also targets context independent specification of software components. It was chosen over annotated UML models, because it specifically models component-based architectures, has standardised semantics, is designed for model-transformations, and involves less complexity. Besides user and component behaviour, this model can describe different types of components, their connections, hardware/software resources, and component allocation to resources in order to enable performance predictions.

Before introducing the new modelling languages for user and component behaviour in Chapter 4, this chapter will first give an overview of basic concepts of the PCM. Chapter 3.1 introduces the Palladio development process model. Chapter 3.2 provides an overview of the PCM meta-model, and describes all parts except the behavioural specification languages. To specify performance properties and parameter dependencies the PCM uses random variables. As they are a prerequisite for the behavioural languages, Chapter 3.3 includes basic concepts of random variables and their realisation in the PCM.

3.1 Palladio Development Process Model

As part of this thesis, the development process model by Cheesman et al. [CD01] (cf. Chapter 2.1.4) has been augmented to explicitly include QoS specification and design time prediction (published in [KH06]) This extended version is also the targeted process model for the Palladio Component Model. The following will describe the participating developer roles (Chapter 3.1.1) and the actual process model (Chapter 3.1.2).

3.1.1 Developer Roles

Early Quality-of-Service (QoS) analysis of a component-based architectures depends on information from different developer roles. The following briefly discusses the responsibilities of the participating roles.

- **Component Developers** are responsible for the specification and implementation of components. They develop components for a market as well as per request. To enable QoS analyses, they need to specify the QoS properties of their components without knowing a) to which other components they are connected, b) on which hardware/software platform they are executed, and c) which parameters are used when calling their services (cf. Chapter 2.4.3). Only such a specification enables independent third party analyses.

- **Software Architects** lead the development process for a component-based application. They design the software architecture and delegate tasks to other involved roles. For the design, they decompose the planned application specification into component specifications. Software architects can select existing component specification from repositories to plan including them into the application. If no existing specification matches the requirements for a planned component, a new component has to be specified abstractly. Software architects can delegate this task to component developers. Additionally, software architects specify component connections thereby creating an *assembly model* (cf. Chapter 3.2.4). After design, software architects are responsible for provisioning components (i.e., decide to make or buy components), assembling component implementations, and directing the tests of the complete application.

- **System Deployers** specify the resources, on which the planned application

shall be deployed. Resources can be hardware resources, such as CPUs, storage devices, network connections etc., as well as software resources, such as thread pools, semaphores, or database connection. The result of this task is a so-called *resource environment specification* (cf. Chapter 3.2.5). With this information, the platform-independent resource demands from the component specifications can be converted into timing values, which are needed for QoS analyses. For example, a component developer may have specified that a certain action of a component service lasts 1000 CPU cycles. From the resource environment specification of the system deployer it would now be known how many cycles the corresponding CPU could execute per second. Besides resource specification, deployers allocate components to resources. This step can also be done during design on the model-level by creating a so-called *allocation model* (cf. Chapter 3.2.5). Later in the development process, during the deployment stage, system deployers are responsible for the installation, configuration, and start up of the application.

- **Domain Experts** participate in requirement analysis, since they have special knowledge of the business domain. They are familiar with the users' work habits and are therefore responsible for analysing and describing the user behaviour. This includes specifying workloads with user arrival rates or user populations and think times. In some cases, these values are already part of the requirement documents. If method parameter values have an influence on the QoS of the system, the domain experts should characterise these values to make predictions more accurate. The outcome of the domain experts' specification task is a so-called *usage model* (cf. Chapter 4.2).

- **QoS Analysts** collect and integrate information from the other roles, extract QoS information from the requirements (e.g., maximal response times for use cases), and perform QoS analyses by using mathematical models or simulation. Furthermore, QoS analysts estimate missing values that are not provided by the other roles. For example, in case of an incomplete component specification, the resource demand of this component has to be estimated. Finally, they assist the software architects to interpret the results of the QoS analyses. It is the goal of component-based QoS prediction methods to automated the tasks of this role as much as possible.

3.1.2 QoS-Driven Development Process

The following integrates the formerly described roles into a component-based development process model featuring QoS analysis (cf. Fig.3.1). The process model inherits the workflows *requirements*, *provisioning*, *assembly*, *test*, and *deployment* from the original model by Cheesman and Daniels [CD01] (described in Chapter 2.1.4). The workflow "specification" has been slightly modified to explicitly include the interaction between component developer and software architect and the specification of extra-functional properties (details follow).

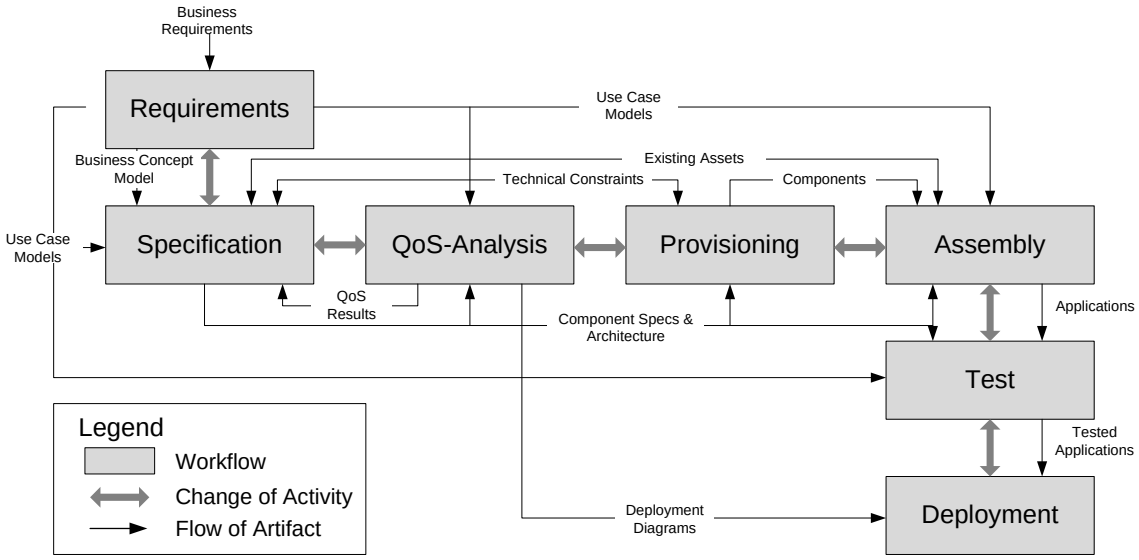


Figure 3.1: QoS-Driven Component-Based Development Process [KH06]

The workflow "QoS Analysis" has been added to the model. Component specifications, the architecture, and use case models are input to the QoS analysis workflow. During this workflow, deployers provide models of the resource environment of the architecture, which contain specifications of extra-functional properties. The domain expert takes the use case models, refines it, and adds QoS-relevant information, thereby creating a complete usage model suitable for QoS predictions. Finally, the QoS-Analyst a) combines all of the models, b) estimates missing values, c) checks the models validity, d) feeds them into QoS predictions tools, and e) prepares a pre-evaluation of their predictions, which is targeted at supporting the design decisions of the software architect. More detail about the QoS analysis workflow follows below. Outputs of the QoS analysis are pre-evaluated results for QoS metrics, which can be used during specification to adjust the architecture, and deployment

diagrams that can be used during deployment to allocate components to different resources.

Specification Workflow The specification workflow (Fig. 3.2, right column) is carried out by the software architect. The workflows of the software architect and the component developers influence each other. Existing components (e.g., from a repository) may have an impact on the inner *component identification* and *component specification* workflow, as the software architect can reuse existing interfaces and specifications. Vice versa, newly specified components by the software architect serve as input for the component requirements analysis of component developers, who design and implement new components.

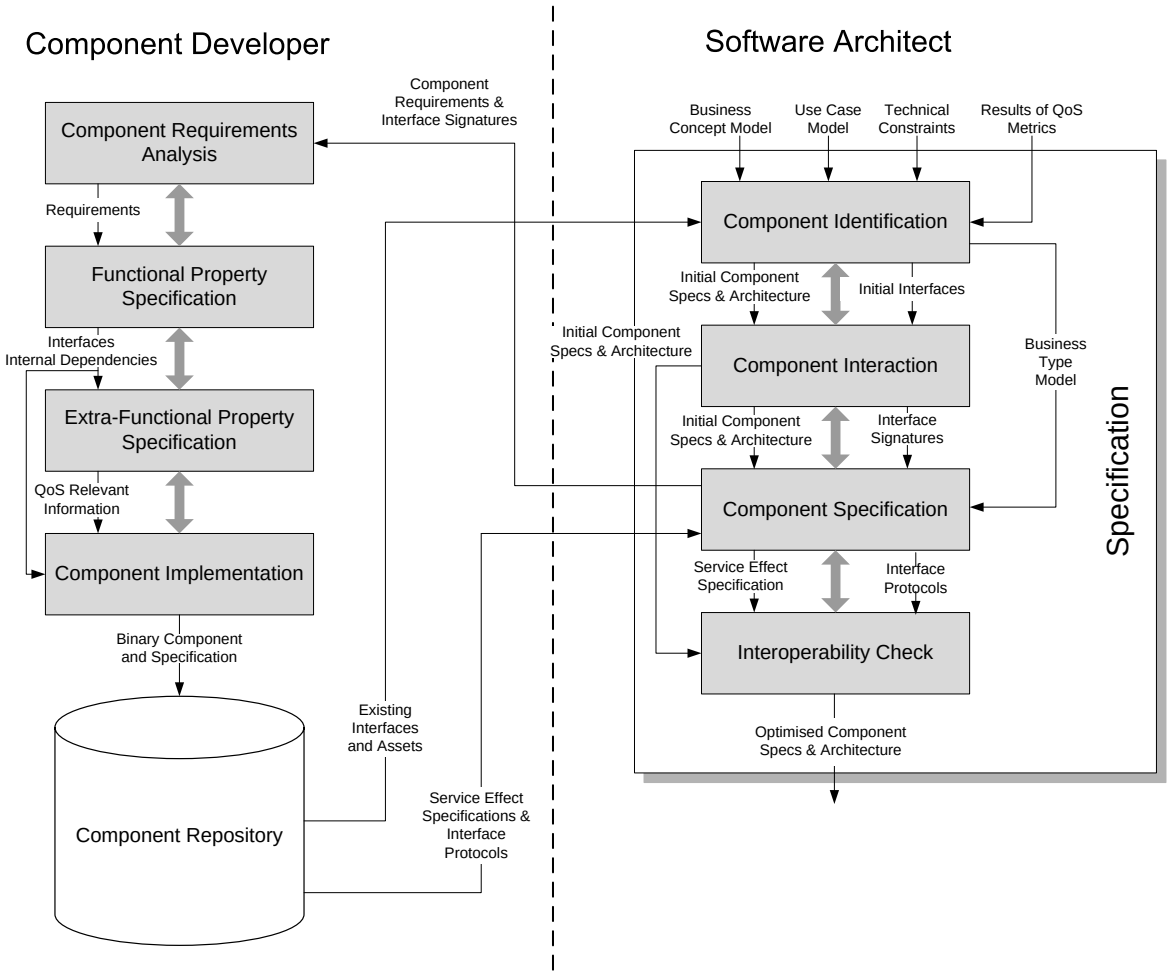


Figure 3.2: Component-Based Development Process: Specification Workflow (Detailed View)

The component developer's workflow is only sketched here, since it is performed separately from the software architect's workflows. If a new component needs to be implemented, the workflow of the component developer (Figure 3.2) can be assumed to be part of the provisioning workflow according to Cheesman and Daniels [CD01].

Any development process model can be used to construct new components as long as functional and extra-functional properties are specified properly. First, component developers have to conduct a *component requirement analysis*. It is succeeded by *functional property specification* and then *extra-functional property specification*. The functional properties consist of interface specifications (i.e., signatures, pre/post-condition, protocols) and descriptions of internal dependencies between provided and required interfaces. Additionally, descriptions of the functionality of component services have to be made. Extra-functional, QoS-relevant information includes for example resource demands, reliability values, and data flow specifications. Finally, after *component implementation* according to the specifications, component developers put the binary implementations and the specifications into repositories, where software architects retrieve and assess them for their architectures.

The specification workflow of the software architect consists of four inner workflows. The first two workflows (*component identification* and *component interaction*) are adapted from [CD01] except they explicitly model the influence on these workflows by existing components. During the *component specification*, the software architect additionally gets existing interface and service effect specifications [Reu01b] as input. Both are transferred to a new workflow called *interoperability check*. In this workflow, interoperability problems are solved and the architecture is optimised. For example, functional parametrised contracts [RS02] can be computed. The outputs of the specification workflow are an optimised architecture and component specifications with refined interfaces.

QoS Analysis Workflow During QoS analysis, the software architecture is refined with information on the deployment context, the usage model, and the internal structure of components. Figure 3.3 shows the process in detail.

The deployer starts with the *system environment specification* based on the software architecture and use case models. Given this information, the required hardware and software resources and their interconnections are derived. As a result, this workflow yields a description of the resource environment, for example, a deployment diagram without allocated components or an instance of the resource environ-

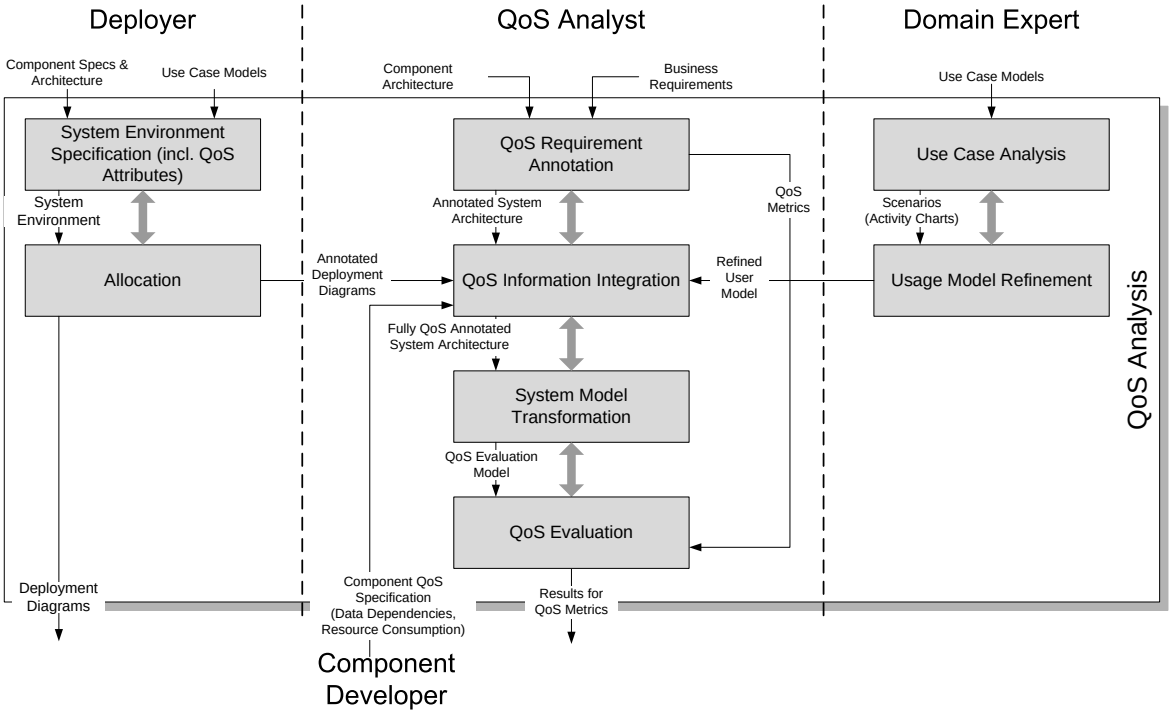


Figure 3.3: Component-Based Development Process: QoS Analysis Workflow (Detailed View)

ment model. Instead of specifying a new resource environment, the deployer can also use the descriptions of existing hardware and software resources. Moreover, a set of representative system environments can be designed if the final resource environment is still unknown. For QoS analysis, detailed information on the resources modelled in the environment are required.

During the *allocation* workflow, the system deployer maps component specifications from the architecture specification to the hardware resources defined in the system environment specification. The result of this workflow is input for the QoS information integration.

The domain expert refines the use case models from the requirements during the *use case analysis* workflow. A description of the scenarios for the users is created based on an external view of the current software architecture. The scenarios describe how users interact with the system and what dependencies exist in the process. Usage models (Chapter 4.2) can be used to describe such scenarios. The scenario descriptions are input to the *usage model refinement*. The domain expert annotates the descriptions with, for example, branching probabilities, expected size of different user groups, expected workload, user think times, and parameter characterisations.

As the central role in QoS analysis, the QoS analyst integrates the QoS relevant information, performs the evaluation, and delivers the feedback to all involved parties. In the *QoS requirement annotation* workflow, the QoS analyst maps QoS requirements to direct requirements of the software architecture. For example, the maximum waiting time of a user becomes the upper limit of the response time of a component's service. For this, the QoS analyst selects QoS metrics, like response time or probability of failure on demand, which are evaluated during later workflows.

During *QoS information integration*, the QoS analyst collects the specifications provided by the component developers, deployers, domain experts, and software architects, checks them for soundness, and integrates them into an overall QoS model of the system. In case of missing specifications, the QoS analyst is responsible for deriving the missing information by contacting the respective roles or by estimation and measurement. The system specification is then automatically transformed into a prediction model (Chapter 6).

The *QoS evaluation* workflow either yields an analytical or simulation result. QoS evaluation aims, for example, at testing the scalability of the architecture and at identifying bottlenecks. The QoS analyst performs an interpretation of the results, comes up with possible design alternatives, and delivers the results to the software

architect. If the results show that the QoS requirements cannot be fulfilled with the current architecture, the software architect has to modify the specifications or renegotiate the requirements.

3.2 PCM Meta-Model

3.2.1 Overview

The formerly described developer roles shall use the Palladio Component Model (PCM) to model their parts of a component-based system. The PCM is a meta-model for specifying component-based software architectures. The Palladio research group has developed the PCM since 2003. Concepts of the model are rooted in Reussner's PhD thesis [Reu01a]. A technical report [RBH+07] provides recent documentation of the PCM's implementation.

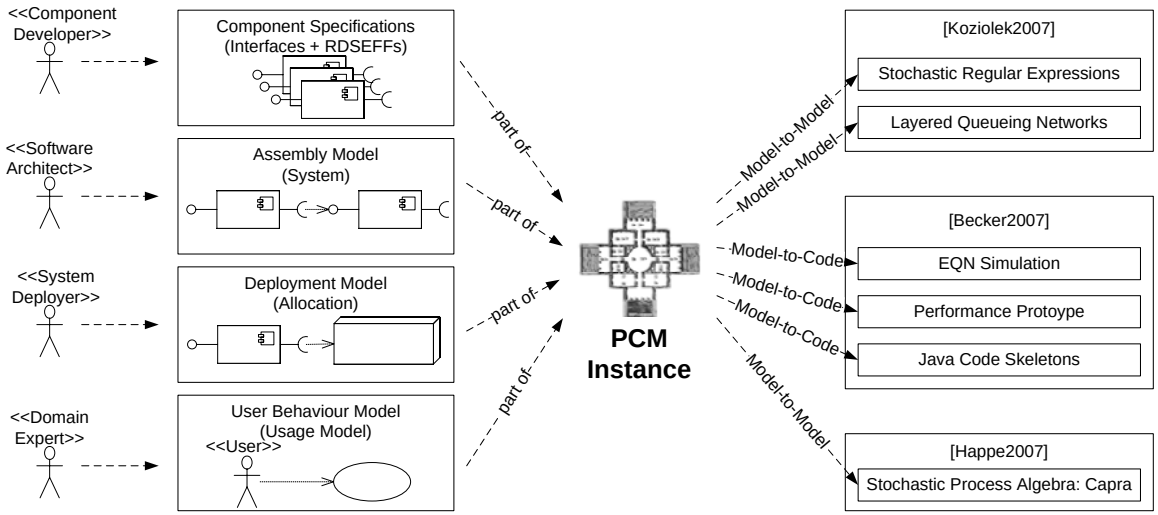


Figure 3.4: Palladio Component Model: Parts and Analysis Models

The PCM meta-model consists of four restricted, domain specific modelling languages, which target four developer roles (Fig. 3.4).

- Component developers specify interfaces, data types, and components.
- Software architects compose the resulting components.
- System deployers model resources and the allocation of components to resources.

- Domain experts describe user behaviour.

Each role uses a modelling language restricted to the concepts of its domain [KH06]. The role concept is aligned with the Palladio development process model introduced before. Component developers can specify components independently from software architects. System deployers only refer to the software architects' system, and may work independently from component developers and domains experts. Domain experts provide usage models independently from system deployers.

After specifying a complete model by combining the individual models from the different developer roles, software architects may use automated model transformations to exploit the model (Fig. 3.4). For performance prediction, model-2-model transformations create stochastic regular expressions (SRE) or queueing networks (QN) from the model. This thesis describes the transformation to SREs (Chapter 6.3) and to LQNs (Chapter 6.4) and solving these formalisms.

Becker's thesis [Bec08] additionally describes three model-2-text transformation of the PCM into an EQN-simulation model for performance predictions, into Java code skeletons for implementation, and into a so-called performance prototype for performance testing. Happe's thesis [Hap08] includes a mapping to a stochastic process algebra named Capra, which enables efficient performance predictions for different multi-user scenarios based on a hybrid approach of analysis and simulation.

Following this overview of the PCM, the next three subsections will provide more detail on several parts of the specification language. Chapter 3.2.2 describes modelling components, interfaces, data types, and repositories. Chapter 3.2.3 introduces the PCM's context model for modelling environmental influences to components. Chapter 3.2.4 shows how to compose components to create composite components and systems. Chapter 3.2.5 explains the PCM's resource modelling and component allocation.

3.2.2 Interfaces, Data Types, and Components

Interface specification and component composition in the PCM are concepts needed to understand the context of the behavioural model for components (RDSEFF) introduced in Chapter 4.3. As elaborated in Chapter 3.1, the PCM targets the following modelling process: Component developers deposit component, interface, and data type specifications (i.e., models) into *repositories*, where software architects can re-

trieve them to build systems, which fulfil specific customer requirements.

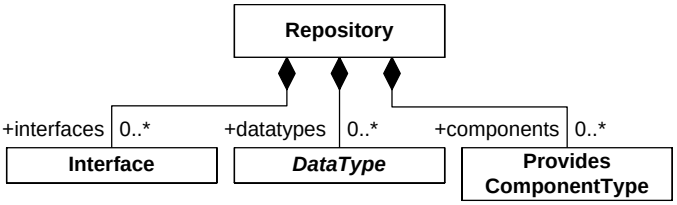


Figure 3.5: PCM Repository (meta-model)

PCM `Repositories` (Fig. 3.5)[1] contain interfaces, data types, and components (as `ProvidesComponentType` in the figure, described later) as first class entities. This means, that these entities may exist on their own without depending on another entity. For example, an interface can be specified in a repository without a component providing or requiring it. A data type can be specified without an interface using it. RDSEFFs are part of the component specification and therefore also included in PCM repositories. However, they are no first-class entities, as they always depend on a particular component specification.

Interfaces & Data Types Component developers can specify PCM interfaces with (i) a list of service signatures and (ii) a protocol constraining valid call sequences. As protocols are not relevant for performance prediction, they will not be described here (details in [Reu01a]).

Signatures in the PCM are rooted in CORBA IDL (Fig. 3.6) [Obj06a]. They contain an ordered list of parameters, each conforming to a specific data type, an unordered list of exception types, and a return type. Parameters have a modifier declaring them as input (`IN`), output (`OUT`), or input/output (`INOUT`) parameters.

Each parameter in a signature conforms to a specific data type. PCM data types are either primitive, collection, or composite types. Several fixed primitive types have been specified (e.g., INT, BOOL, STRING, DOUBLE). The PCM interface definition language does not restrict developers to a specific programming language, but provides common data types in a repository so that different component developers can refer to the same types. Collection data types model sets of elements of a specific data type. They may be used to model data structures such as arrays, bags, lists, trees, hash maps, etc. They contain an inner type, so that the elements

[1]Note that the classes in this diagram are meta-classes from the PCM. The `EClass` stereotype of the ECore language has been omitted for clarity.

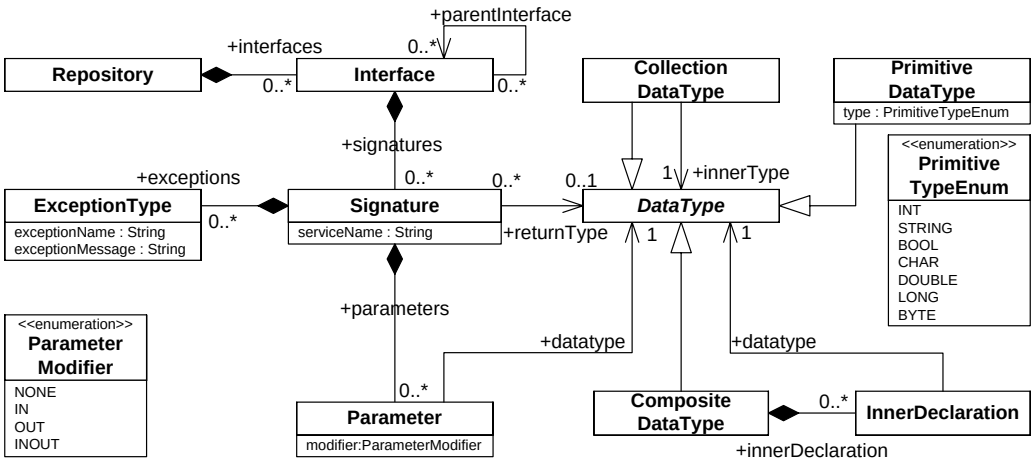


Figure 3.6: Interface (meta-model)

contained in the collection have a common data type. Composite data types consist of an arbitrary number of inner elements each conforming to a different data type. Records, structs, etc. can be modelled with composite data types.

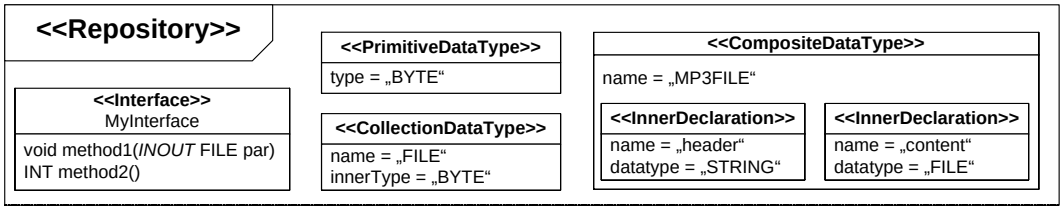


Figure 3.7: Example Instances: Interface, Data types

An example repository helps to illustrate PCM interfaces and data types (Fig.3.7)[2]. The interface MyInterface exists on its own and is so far not connected to a particular component. It contains two services (method1, method2), which specify input parameters and return values. The primitive data type BYTE uses the pre-specified BYTE to define a data type. The collection data type FILE contains this primitive data type as its inner elements and is thus an array of bytes. Finally, the collection data type MP3FILE includes two inner declarations: a ”header” of the data type STRING and a ”content” of the formerly specified collection data type FILE.

The QoS properties of a service are not part of the interface specification, as they

[2]The diagram shows an instance of the PCM meta-model with UML as the concrete syntax. Classes are stereotyped with the corresponding PCM meta-classes they instantiate.

depend on a particular component implementing the service, while the interface must remain independent from component implementation. Component developers can model them as part of components with RDSEFFs (Chapter 4.3).

Components Components are black-box entities with contractually specified interfaces [SGM02]. Therefore, components are mainly specified by their relation to interfaces, which is described in the following. Components do not contain interfaces, as interfaces exist on their own and are per se not part of components. A component (`ProvidesComponentType`) is an `InterfaceProvidingRequiringEntity` (Fig. 3.8), which contains a set of so-called *roles* (`ProvidedRole` or `RequiredRole`).

If a component contains a `ProvidedRole` referring to an interface I, it means it implements the interface. If the component's environment provides all required services of this implementation, the component can provide the services specified in interface I. Clients can call the component for the specified services. A `RequiredRole` within a component can have different meanings depending on the component type, as described further below. For basic components, it means the component requires the services in the interface referenced by the role, as its own service implementations call these services.

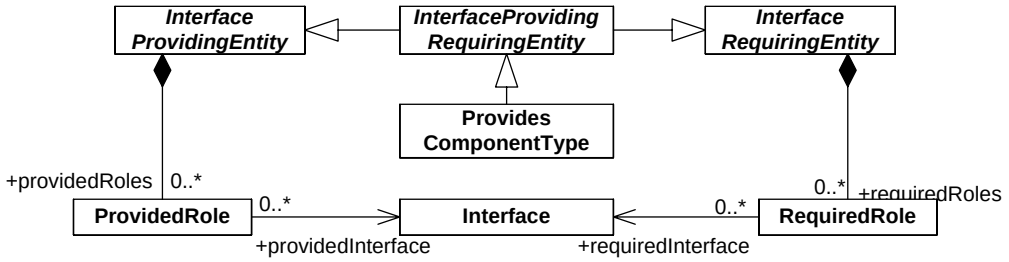


Figure 3.8: Component, Role, Interface (meta-model)

The PCM supports three types of components (namely `ProvidesComponentType`, `CompleteComponentType`, `ImplementationComponentType`), which differ in the obligation to their roles (Fig. 3.9). The different types reflect different stages in a component's development life cycle. They include different kinds of usage profile dependent QoS specifications and are therefore described in the following.

For components of a `ProvidesComponentType`, only `ProvidedRoles` are mandatory, while `RequiredRoles` are optional. A component of a `Provides-`

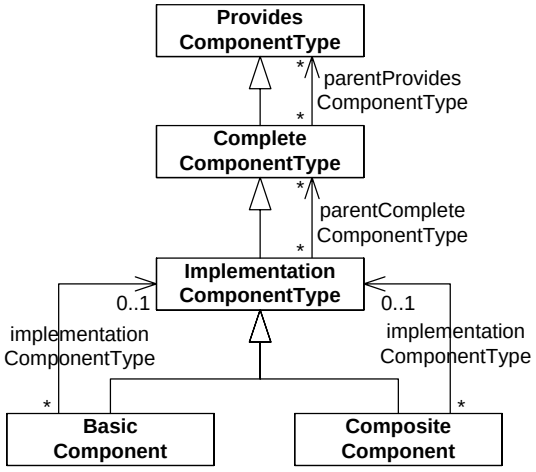


Figure 3.9: Component Types (meta-model)

`ComponentType` can, but does not need to contain `RequiredRoles`. For a developer implementing the component it is not mandatory to use the referenced services to stay type conform. The component developer may also use additional required services. This models the knowledge during a very early development stage of a software component, when it is known that a certain functionality is needed, but it is still unclear which other components are needed to fulfill this functionality.

To specify the performance of a `ProvidesComponentType` component, the component developer can make no references to external services or the internal behaviour as they are still unknown. Therefore, the component developer cannot provide a behavioural specification such as a RDSEFF of the component's services. Instead, a `ProvidesComponentType` component can contain a single `QoS-Annotation`, which only specifies an execution time and may only depend on the input parameters of the provided services.

The obligation to roles changes for `CompleteComponentTypes`. They contain mandatory `ProvidedRoles` and mandatory `RequiredRoles`: A component developer implementing this type of component may not use more required roles than specified by the `CompleteComponentType` to remain type conform. The dependencies between provided and required roles remain unspecified and may differ in different component implementations. Software architects may give `Complete-ComponentType` as a requirements specification to component developers. The performance specification of a `CompleteComponentType` component is a `QoS-Annotation` like for the `ProvidesComponentType`.

ImplementationComponentTypes are CompleteComponentTypes, but additionally fix the dependencies between provided and required roles with a service effect specification (details in [Reu01a]). This type describes a particular implementation and is further subdivided into BasicComponents and CompositeComponents.

BasicComponents are atomic building blocks and cannot be further decomposed into subcomponents. They encapsulate their content and contain an abstract behavioural description (RDSEFF, see Chapter 4.3) for each provided service to specify its performance.

Component developers can build CompositeComponents by composing BasicComponents or other CompositeComponents (see Chapter 3.2.4). They are containers of other components and do not provide any additional functionality on their own. A CompositeComponent's performance specification can be derived by composing the inner components' RDSEFFs.

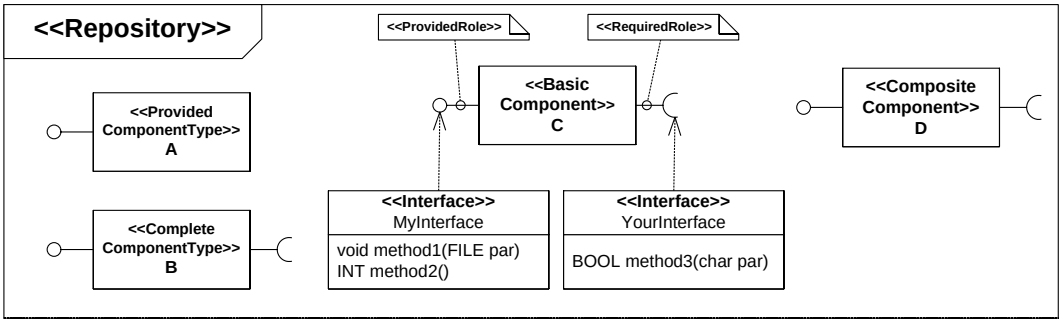


Figure 3.10: Example Instances: Components

As an example, the repository in Fig. 3.10 contains a basic component C, which is connected to the interface MyInterface in a providing role and to the interface YourInterface in a requiring role. The repository also contains a still unfinished component A of a provided component type, which does not specify required services. The complete component type component B includes connections to provided and required interfaces. The component needs to implement the services specified in the provided interface and use only the services specified in the required interface along the way. Finally, the composite component D is composed out of inner components, which are encapsulated within it and are not visible to a software architect or other component developer composing the component.

3.2.3 Context Model

The PCM features a so-called context model, which includes information about a component's binding, allocation, and usage. While component developers supply individual component specification, further information about each component in an architecture is necessary for QoS predictions. This information is only available after component implementation and cannot be supplied by the component developer. Therefore, the PCM allows separate creation of the context model by other developer roles or tools.

Before describing the context-model's implementation in the PCM, first the general concept shall be explained. The context-model includes manually specifiable and computable parts (Tab. 3.1).

	Assembly Context	Allocation Context	Usage Context
Specified	• Horizontal Composition: Binding to other Components • Vertical Composition: Encapsulation in Composite Components	• Allocation to Hardware Resources • Configuration ○ Component ○ Container ○ Concurrency ○ Communication ○ Security ○ ...	• Usage at System Boundaries ○ User Arrival Rate ○ Number of Users ○ Request Probabilities ○ Parameter Values
Computed	• Parametric Contracts ○ Provided/Required Services ○ Provided/Required Protocols ○ ...	• Allocation-dependent QoS Characteristics ○ Timing Values for Resource Demands ○ ...	• Usage inside Components ○ Branch Probabilities ○ Loop Iteration Numbers ○ Input/Output Parameters

Table 3.1: Component Context relevant for QoS Prediction

The assembly context refers to a component's binding to other components. The manually specifiable part includes both the connections to other components via provided and required interfaces and the containment relationship between vertically composed components. The computable part refers to the parametric contracts introduced by Reussner [Reu01a]. For example, they allow to restrict the set of required services of a component if certain provided services are not needed. A `BasicComponent` can have multiple assembly contexts in the same architecture. In this case, each assembly context refers to a copy of the same component implementation.

The allocation context refers to a component's binding to hardware/software resources. It requires manual specifications of a component's allocation and con-

figuration options related to hardware and software resources. Tools can compute allocation-dependent QoS characteristics of a component by combining information from the component specification and the hardware environment. For example, a resource demand provided by a component developer can be transformed into a timing value, if the speed of the underlying hardware resource is known. A `BasicComponent` can have multiple allocation context in the same architecture, in which case each allocation context refers to a copy of the same component implementation running on different resources.

The usage context refers to a component instance's usage by clients. For PCM instances, only the user behaviour at the system boundaries needs to be specified. This includes the number of users, which services they call, and what parameter values they supply. Tools can then propagate these values to each component specification in the architecture and compute individual component usage including branch probabilities, loop iteration numbers, and parameter values. Chapter 6.2 explains this in detail.

Tab. 3.2 depicts the specification responsibilities for the manually specified parts of the context-model and includes references to descriptions of the respective parts in the PCM meta-model in this thesis. The computable parts of the context-model are not part of the PCM meta-model, as they depend on the desired analysis method. In this thesis, the so-called Dependency Solver (Chapter 6.2) performs the necessary computation of allocation context and usage context information. Notice that the computation of parametric contracts is currently not implemented.

	Assembly Context	Allocation Context	Usage Context
Specified	• Specification by Software Architect • Section 3.2.4	• Specification by System Deployer • Section 3.2.5	• Specification by Domain Expert (Usage Model) • Section 4.2
Computed	• No Implementation Available • cf. [140]	• Computed by Dependency Solver • Section 6.2	• Computed by Dependency Solver • Section 6.2

Table 3.2: Context Model Implementation in the PCM

3.2.4 Composition

Software architects compose components to build systems, and component developers compose components to build composite components. Therefore, the PCM has a unified concept of design-time composition, as both roles specify a so-called

ComposedStructure (Fig 3.11). Software architects build a System model as a special ComposedStructure, which defines the boundaries of the system under study. A System contains provided and required roles as it is an InterfaceProvidingRequiringEntity. However, system ProvidedRoles can only be used by UsageModels (Chapter 4.2), but not by other components. In the same manner, system RequiredRoles do not use other components explicitly, as it is assumed that these components lie outside the system boundaries (e.g., web services). It is however possible to specify an input-parameter dependent time-consumption for such services. Opposed to this, the ProvidedRoles and RequiredRoles of CompositeComponents can connect to other components.

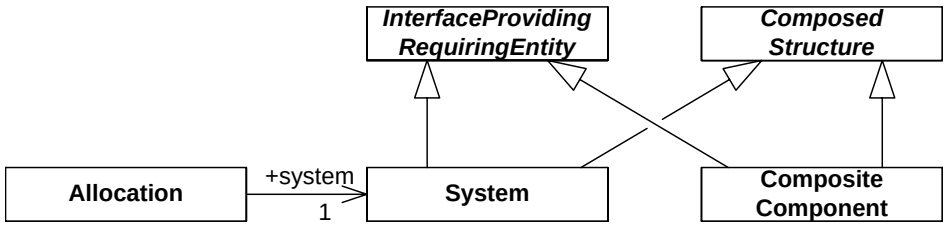


Figure 3.11: System and Composite Component (meta-model)

A ComposedStructure (Fig. 3.12) stores the interconnection between a set of components. It does not directly contain components, but AssemblyContexts, which reference components. With this indirection, AssemblyContexts allow using multiple deployment instances (i.e., not runtime instances) of the same component type within the same system. A ComposedStructure contains a number of AssemblyConnectors to bind the roles of two component instances, each referencing a providing assembly context and a requiring assembly context. Additionally, they reference the involved ProvidedRole and RequiredRole, which they connect.

As ComposedStructures do not include additional functionality besides the functionality of their embedded components, they have to delegate calls to their interfaces to interfaces of embedded components. In the same manner, calls of their embedded components to required services have to be delegated to the ComposedStructures required interfaces. Therefore, ComposedStructures contain a number of ProvidedDelegationConnectors and RequiredDelegationConnectors, which connect outer roles with inner roles.

Consider the composite component A in Fig 3.13, which is a composed struc-

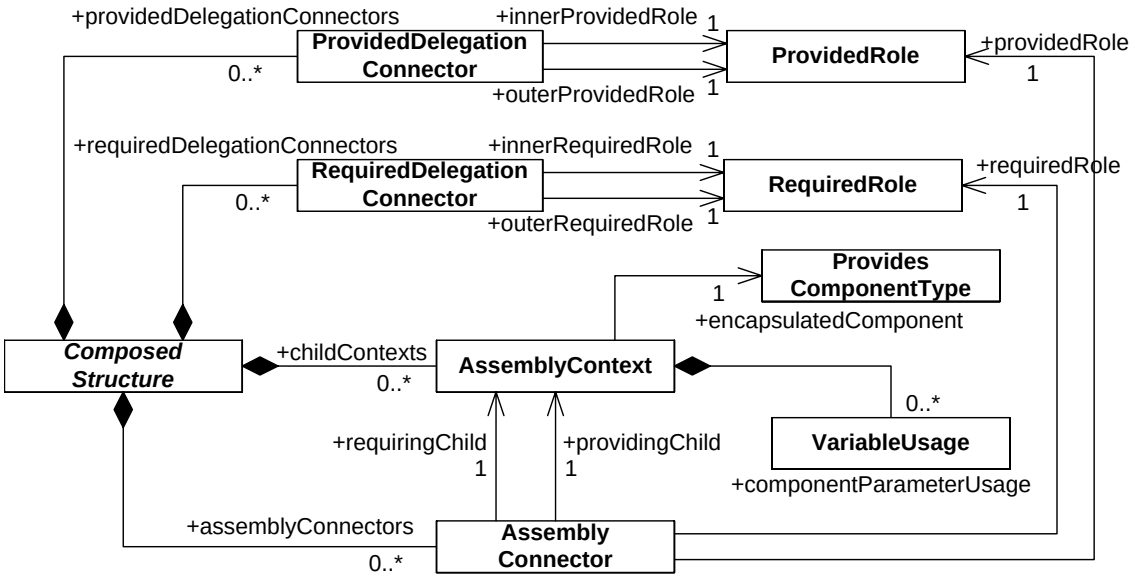


Figure 3.12: Composed Structure (meta-model)

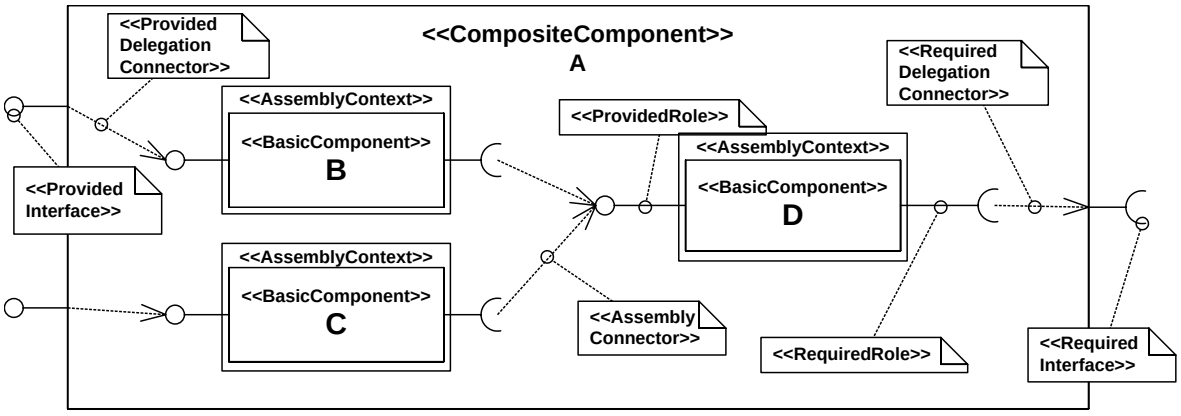


Figure 3.13: Example Instances: Composite Component

ture. It contains three assembly contexts referencing the components B, C, and D. The provided interfaces of the composite component are bound to inner provided interfaces with provided delegation connectors, while the required interface of the embedded component D is bound to the composite component's required interface via a required delegation connector. The required role of B is connected to the provide role of D through an assembly connector. Furthermore, the required role of C is also connected to the provided role of D with an assembly connector.

3.2.5 Resource Model, Allocation

For performance prediction it is essential to model the resource environment of a system, as the resource are a major influence factor on the perceived responsiveness and throughput of an application. A PCM `System` containing a set of assembled components models only the software side of an application, and needs to be augmented with a resource environment model and the allocation of components to resources.

In UML, deployment diagrams provide means to specify component deployment. In the PCM, resource modelling and component allocation (i.e., assigning components to resources during design time) are tasks of the system deployer role. Therefore, the PCM provides a domain specific language for the system deployer. It allows modelling resources and allocating components to these resources. This language includes resource attributes such as processing rates of processors or latencies of network resources. It enables software architects to analyse the responsiveness of their system for resources with different attributes and to answer sizing questions (i.e., what hardware is needed to ensure a certain performance?).

In CBSE, resource modelling differs from resource modelling in monolithic architectures, because of the independent component specification by different developers. When component developers specify the performance of their components, they can and should not know the concrete resource instance a component will be using to keep their specification independent from a specific context. Thus, the PCM introduces the concept of abstract `ResourceTypes`, which component developers reference when specifying resource demands in their RDSEFFs (Chapter 4.3). A `ResourceType` represents a class of resources (e.g., CPU, hard disk, LAN connection) without specifying concrete properties of the resources (e.g., processing rate, throughput, latency), which are unknown during component design time.

`ResourceTypes` reside in so-called `ResourceRepositories` (Fig. 3.14). A

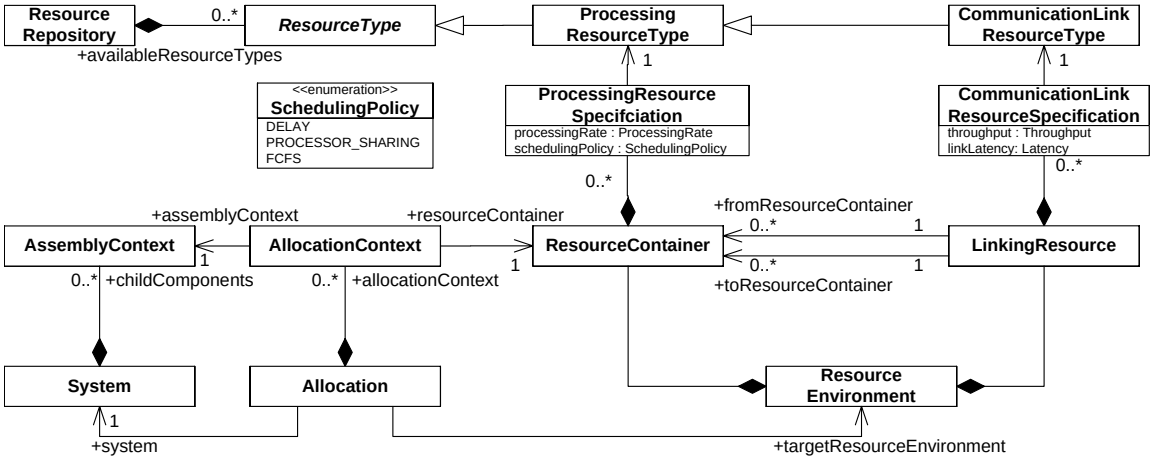


Figure 3.14: Resource Model, Allocation (meta-model)

fixed resource type repository is needed for the specification of resource demands, so that different component developers refer to the same resource classes. The PCM currently supports two kinds of `ResourceTypes`: `ProcessingResourceTypes` to model CPUs and storage devices, and `CommunicationLinkResourceTypes` to model network connections. Additionally, the PCM provides a standard minimal `ResourceRepository`, which contains CPU, HD, and LAN `ResourceTypes`. Component developer must use this repository when referencing resources in their RDSEFFs.

Besides the abstract resource type specification, the PCM allows a concrete `ResourceEnvironment` specification for system deployers (Fig. 3.14). A `ResourceEnvironment` consist of a number of `ResourceContainers` representing computers and a number of `LinkingResources` representing network connections between `ResourceContainers`.

Each `ResourceContainer` may contain a number of concrete resources (i.e., instances of resource types). These resource type instances are called `Processing-ResourceSpecification` in the PCM. They are attributed with a processing rate (e.g., CPU-cycles/sec, bytes read/sec, etc.), and a `SchedulingPolicy`. The PCM supports delay, first come first serve (FCFS), and processor sharing (PS) scheduling (also see [LZGS84]). A resource with delay scheduling does not possess a queue and processes each request instantaneously. Therefore, delay resource do not cause contention delays. Resources with FCFS scheduling have a queue for each incoming request, and process them in the order of their arrival. Processor sharing is an

idealised form of round robin scheduling with zero context switch times and unlimitedly small time slices. For example, a system deployer can represent a CPU with a PS scheduling policy and a processing rate of 3 billion cycles per second, and a hard disk with a FCFS scheduling policy and a processing rate of 30 MB/sec.

Notice, that a `ResourceContainer` can contain at most one resource instance per resource type (e.g., one CPU instance per CPU type). This is necessary, so that behavioural component specifications can reference resource types unambiguously. It is however possible to specify multi core CPU types.

Each `LinkingResource` connecting `ResourceContainers` may include several `CommunicationLinkResourceSpecifications`. These reference `CommuncationLinkResourceTypes` and are attributed with a latency and a throughput. The latency specifies the round trip delay for a network packet on the link (e.g., 200 ms). The throughput specifies the number of bytes, which can be transferred through the link in a second (e.g., 100 Gigabit). For example, a system deployer may specify a Gigabit-LAN connection with a latency of 0.5 ms and a throughput of 1000 Gigabit.

System deployers can model new resource environments, which are not yet realised in hardware, or model existing resource environments to check whether a component-based architecture could be deployed into a legacy environment while still providing adequate performance. So far, the PCM `ResourceEnvironment` only includes hardware resources. This will be improved in the future to also represent middleware resources and performance-relevant characteristics of operating systems.

With a `ResourceEnvironment` specification and a `System` provided by the software architect, the system deployer can allocate components from the `System` to `ResourceContainers` within the `ResourceEnvironment`. A PCM `Allocation` (Fig. 3.14) consists of a number of `AllocationContexts`, which establish the connection between an `AssemblyContext` and a `ResourceContainer`. Providing an `AllocationContext` for a component instance embedded in an `AssemblyContext` means that the component accesses the `ProcessingResourceSpecifications` within the referenced `ResourceContainer`. Because the `ResourceTypes` are uniquely referenced within a `ResourceContainer`, tools can determine the concrete `ProcessingResourceSpecification` for a specific `ResourceType` referenced by the component's RDSEFF.

The example in Fig. 3.15 contains two resource containers including a CPU and a HD, and a CPU respectively. The containers are connected via a FastEthernet

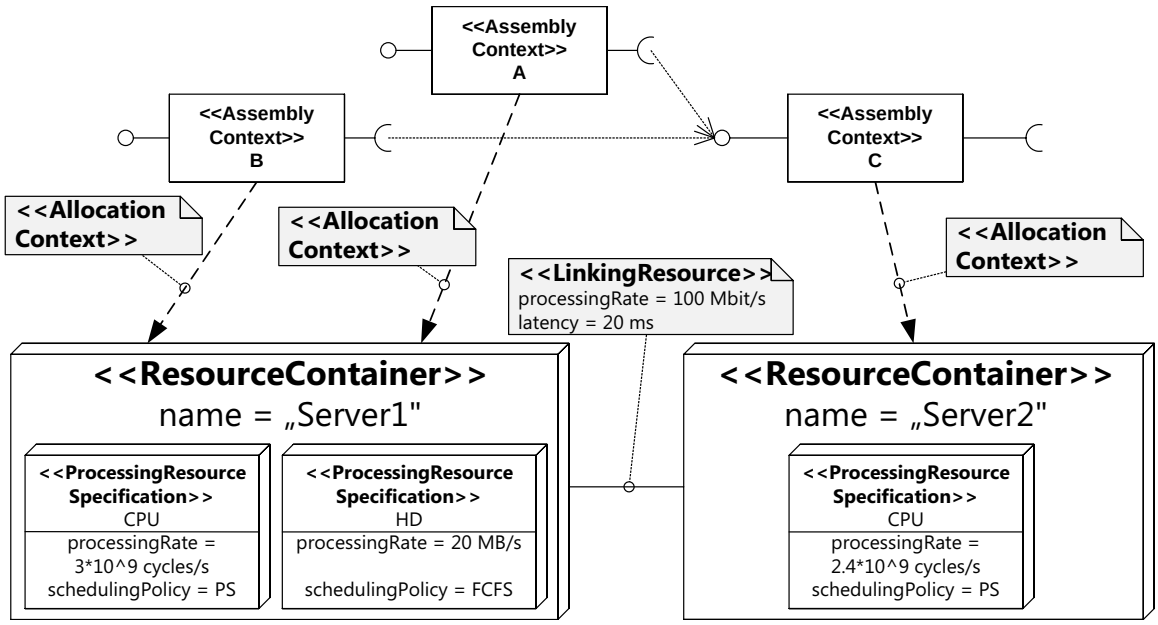


Figure 3.15: Example Instance: Resource Environment, Allocation

connection. The system deployer has allocated the components A and B to Server1 and the component C to Server2.

3.3 Random Variables

3.3.1 Motivation

The modelling languages introduced in the following subsections use *random variables* to describe user and component behaviour. Random variables allow not only constant values (e.g., 3 loop iterations), but also probabilistic values (e.g., 2 loop iterations with a probability of 0.4 and 3 loop iterations with a probability of 0.6). They are well-suited for capturing uncertainty when modelling systems during early development stages. The following motivates the necessity and usefulness of modelling with random variables, because of the uncertainty of user behaviour, component behaviour, resource demands, and execution environment.

Domain experts can only approximate expected *user behaviour* when creating usage models. For the domain of distributed and web-based business information systems, the exact number of customers using the system at a given point in time is usually unknown. Domain experts can describe them with statistical means. Fur-

thermore, they can model the behaviour of users in terms of invoked component services and input parameters more detailed with random variables instead of less expressive constant values.

User behaviour influences *component behaviour*, which therefore is also subject to uncertainty. User input propagates through component-based software architectures and influences the control and data flow between components. A service invoked by a user may call other components with different parameter values depending on its own parameter values or values returned by other components. For example, a component service might call a required service in a loop, whose number of iterations depends on the size of a collection it receives as an input parameter. A larger number of concurrent calls increases the perceived response time of a service because of contention delays.

Besides this uncertainty introduced by user behaviour, component developers often cannot specify the execution times of software components with certainty as explained in the following. The RDSEFF language introduced in Chapter 4.3, which allows specifying *resource demands* of component services, abstracts from the source code of a software component. It potentially combines a large number of program statements into a single resource demand specification, whose amount is therefore not exact, but uncertain. It is more accurately described by a random variable instead of a constant value. The abstraction from program code is necessary to keep the model analysable and to retain the component developer's intellectual properties, who does not want to expose the algorithms used in the implementation.

Another factor introducing uncertainty into the models is the components' *execution environment*, which spans middleware, application servers, virtual machines, operating systems, and hardware. For performance predictions, the execution environment is a major influencing factor on the perceived response times and throughput of a component-based system [LFG05]. Developers might not know several properties of the execution environment during early development stages (e.g., size of thread pools, type of application server, hardware configuration, etc.). Other properties are even non-deterministic and occur on hardly predictable occasions (e.g., garbage collection, forced component restarts, etc.). The targeted domain of distributed systems usually does not use real-time execution platforms, therefore there are no hard upper bounds for execution times of components.

For these four performance-relevant factors, which are subject to uncertainty during early development stages, it is useful to express performance annotations in component models with random variables. Such annotations include but are not

limited to loop iteration number, resource demands, parameter values, user arrival rates, etc.

The following subsections define random variables and describe their use in the PCM, where they have been packaged in the so-called Stochastic Expression (StoEx) framework. Chapter 3.3.2 provides a formal definition of random variables and their probability distribution. Chapter 3.3.3 briefly shows different supported data types for constant random variables, before Chapter 3.3.4 describes the use of discrete random variables in the StoEx-framework. Chapter 3.3.5 deals with continuous random variables and their realisation. They are useful to model timing values. Finally, Chapter 3.3.6 presents the syntax and semantics of the expression language underlying the StoEx-framework, which allows boolean, compare, and arithmetic operations with random variables.

3.3.2 Basic Properties of Random Variables

Informally, a random variable maps outcomes of a random experiment to values. For example, rolling a dice twice and adding the results yields a random variable with the range $\{2, 3, ..., 12\}$. Mathematically, a random variable (Appendix B.1.3) is a measurable function X (Appendix B.1.2) from a probability space (Appendix B.1.1) into a measure space (Appendix B.1.1). Typically, real-valued random variables are used (Definition 14).

Definition 14 Real-valued Random Variable

Let $(\Omega, \mathcal{A}, P)$ be a probability space and $(\mathbb{R}, \mathcal{A}')$ be a measure space with $\mathcal{A}'$ being the Borel σ-algebra. An $(\mathcal{A}, \mathcal{A}')$-measurable function $X : \Omega \to \mathbb{R}$ is a ***real-valued random variable*** mapping a real number $X(\omega)$ to each element $\omega \in \Omega$, if

$$\forall r \in \mathbb{R} : \{\omega | X(\omega) \leqslant r\} \in \mathcal{A}$$

meaning that the set of all results below a certain value must be an event.

The results of $X(\omega)$ (i.e., the real numbers, for example, in a coin toss, a 0 for heads and 1 for tails) are often not important, and are therefore sometimes not presented explicitly. More interesting is the probability distribution $P_X(A) : \mathbb{R} \to [0, 1]$ induced on the range of X by the probability measure P of the probability space $(\Omega, \mathcal{A}, P)$. It is given by

$$P_X(A') = P(X^{-1}(A')), \quad \forall A' \in \mathcal{A}'$$

which is the probability measure P for the inverse image of the outcomes of X.

This induced distribution gives probabilities for the outcomes of a random experiment. For example, when rolling a dice twice, the probability of getting a 2 as the sum of results is $P(X = 2)$[3]$= P(1) * P(1) = \frac{1}{6} * \frac{1}{6} = \frac{1}{36}$.

The cumulative distribution function (CDF) completely describes the probability distribution of a real-valued random variable:

Definition 15 Cumulative Distribution Function

Let $(\Omega, \mathcal{A}, P)$ be a probability space, $X : \Omega \to \mathbb{R}$ be a real-valued random variable, and $t \in \mathbb{R}$. The function $F_X(t) : \mathbb{R} \to [0..1]$ with

$$\begin{aligned} F_X(t) &= P(\omega \in \Omega | X(\omega) \leqslant t) \\ &= P(X \leqslant t) \end{aligned}$$

is called **cumulative distribution function** or the probability distribution function of X.

If the CDF of a random variable is known, it is possible to compute the probability, that the random variables takes values between two real numbers:

$$P(a < X \leqslant b) = P(X \leqslant b) - P(X \leqslant a) = F_X(b) - F_X(a)$$

For example, when throwing a single dice, the probability of getting a number between 3 and 5 is $P(2 < X \leqslant 5) = F(5) - F(2) = \frac{5}{6} - \frac{2}{6} = \frac{3}{6} = \frac{1}{2}$.

The StoEx-framework supports constant, discrete, and continuous random variables, which will be described in the following.

3.3.3 Constant Random Variables in the StoEx-framework

A constant random variable adopts only a single value (i.e., $X(\omega) = c, \forall \omega \in \Omega$). It is a special case of a discrete random variable. The PCM supports the following types of constant random variables:

- integer, named `IntLiteral`[4]
- real, named `DoubleLiteral`
- boolean, named `BoolLiteral`

[3] $P(X = r)$ is a short-hand notation for $P(\{\omega \in \Omega : X(\omega) = r\})$ for some $r \in \mathbb{R}$.

[4] Refer to Fig. 3.21 for the realisation of these types in the StoEx-framework meta-model.

- enum, named `EnumLiteral`

Developers can use constant random variables to assign fixed values to parameters of their models (e.g., the number of loop iterations executed by a component service, user input parameters, user population, etc.), although these are not able to express uncertainty and might make the models less accurate.

3.3.4 Discrete Random Variables in the StoEx-framework

A random variable is called discrete, if it only adopts a finite or countable infinite set of values. For example, the number of loop iterations and the population of users in a closed workload are expressed as discrete random variables in the PCM. Non-constant, discrete random variable adopt different values and allow more expressive modelling than constant random variables.

Their probability distribution is a probability mass function (pmf). A pmf gives the probability that a discrete random variable (with a finite or countable infinite sample space) is exactly equal to some value:

Definition 16 Probability Mass Function

Let $(\Omega, \mathcal{A}, P)$ be a probability space and X be a discrete random variable taking values on a countable sample space $S \subset \mathbb{R}$. Then the probability mass function $p_X : \mathbb{R} \to [0, 1]$ of X is given by

$$p_X(x) = \begin{cases} P(X = x) = \sum_{X(\omega)=r} P(\omega), & x \in S, r \in \mathbb{R}, \omega \in \Omega \\ 0, & x \in \mathbb{R} \backslash S \end{cases}$$

This defines $p_X(x)$ for all real values, including the ones, x can never adopt. Their probability is always zero.

The StoEx-framework supports the following event spaces Ω for discrete random variables:

- $\mathbb{N}$, for integer values, with a pmf denoted as `IntPMF`
- $\mathbb{R}$, for real values, with a pmf denoted as`DoublePMF`
- $\mathbb{B}$, for boolean values, with a pmf denoted as `BooleanPMF`
- $\mathbb{E}_n$, for enumerations, with a pmf denoted as `EnumPMF`

Random variables map these event spaces to real-valued measure spaces. In the context of the StoEx-framework this actual mapping is not important, as PCM analysis and simulation tools only need the corresponding pmfs for their computations involved in performance predictions. Therefore, the StoEx-framework only stores the pmfs of discrete random variables (Fig. 3.16). It is an ordered list or an unordered set of `Samples`. A `Sample` has a value representing an element from the sample space (of type Integer, Double, Boolean, or Enumeration) and a probability.

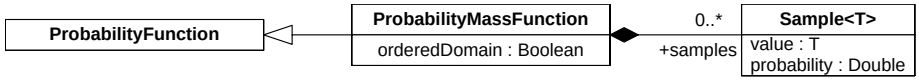
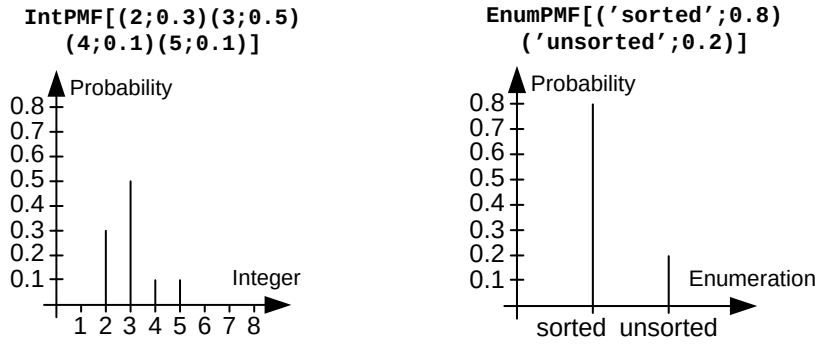


Figure 3.16: Probability Mass Functions (meta-model)

Furthermore, the StoEx-framework defines a concrete textual syntax for instances of `ProbabilityMassFunctions` and implements a lexer and parser for it. The textual syntax offers a convenient way for developers to enter `ProbabilityMassFunctions` into tools. It first uses the kind of the pmf (`IntPMF`, `DoublePMF`, etc.) as a keyword, and then includes a list of value/probability pairs enclosed in parenthesis.



(a) Integer pmf

(b) Enumeration pmf

Figure 3.17: Examples for pmfs in the StoEx-framework

Fig. 3.17 depicts two examples. Each contains the textual syntax of the pmfs at the top and a corresponding graphical syntax at the bottom. The first pmf (Fig. 3.17(a)) is of type integer, and for example models the number of loop iterations in a component behaviour specification, or the characterisation of values of some integer parameter. The second pmf (Fig. 3.17(b)) is of type enumeration, and

characterises the structure of a collection data type as sorted (80 percent probability) or unsorted (20 percent probability).

The StoEx-framework does not restrict developers to modelling special discrete distributions like geometrical or binomial distributions. Instead, it allows modelling any kind of discrete distribution. This is useful, as discrete probability distributions in distributed systems often do not follow standard distributions. Therefore, approximating such distributions with standard distributions might lead to inaccurate performance predictions. A specific example of such a situation follows in Chapter 4.3.

3.3.5 Continuous Random Variables in the StoEx-framework

A random variable is called continuous, if it adopts an uncountable infinite number of values. The probability distribution of a continuous random variable is given by a probability density function (pdf). A pdf represents a probability distribution in terms of intervals:

Definition 17 Probability Density Function

A non-negative Lebesgue-integrable function $f_X : \mathbb{R} \rightarrow \mathbb{R}$ is called ***probability density function*** of the random variable X, if

$$\int_a^b f_X(x)dx = P(a \leqslant X \leqslant b)$$

for any two numbers a and b, $a < b$. The total integral of $f_X(x)$ has to be 1 (i.e., $\int_{-\infty}^{\infty} f_X(x)dx = 1$).

The probabilities of single results for a continuous random variable are zero for each result. For a pdf $f(x)$, it is only possible to give probabilities for an interval $[a, b]$ around x. The pdf definition states that the probability of a continuous random variable X taking values between a and b is the integral of $f(x)$ with the limits a and b.

In the PCM context, developers may use continuous random variables to model timing values and resource demands. For example, they do not specify the probability for an execution time being exactly 2.0 seconds long, but its probability for the time being between 2.0 and 3.0 seconds. This is adequate, because the probability of an execution time being exactly 2.0 seconds with an arbitrary number of decimal places is theoretically and practically zero.

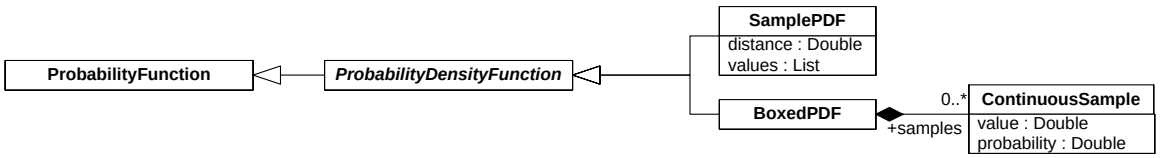


Figure 3.18: Probability Density Functions (meta-model)

Standard pdfs include for example uniform and normal distributions. To not restrict developers to these pdfs with standard distributions, it is desirable to support non-standard pdfs. For these pdfs, it is hard to find a closed form (i.e., a formula describing the pdf). Therefore, the StoEx-framework handles pdfs in a discretised form. Instead of storing the probability density, the StoEx-framework stores the probabilities of intervals of the pdf, which are given by the intervals' integrals. There are different ways to determine appropriate interval sizes, and the StoEx-framework realises two of them (Fig. 3.18). `SamplePDFs` use a fixed interval size called "distance" and store a list of probabilities each for a single interval. `BoxedPDFs` use a variable interval size and store a list of samples each with the right limit of the interval and the interval's probability. The following explains the rational behind the two variants.

Sampled Probability Density Function `SamplePDFs` partition a pdf's domain into $N \in \mathbb{N}$ intervals denoted by the set I. Each interval has the same width $d \in \mathbb{R}^+$ (for "distance"). The interval $[(i-\frac{1}{2})d, (i+\frac{1}{2})d[$ defines the ith element ($i \in 1, ..., N$) in I. Assuming a pdf's domain always being greater or equal to zero, the first interval ($i = 0$) is set to $[0, \frac{1}{2}d[$. The probability p_i associated to each interval i is its mean value $i * d$, which minimises computational errors from the discretisation. Therefore, a `SamplePDF` stores a set of N probabilities with p_i defined as:

$$p_i = \int_{(i-\frac{1}{2})*d}^{b} f(x)dx, \quad lim_{b \to (i+\frac{1}{2})*d}, \quad \text{for } i > 0, \quad \text{and}$$

$$p_i = \int_{0}^{b} f(x)dx, \quad lim_{b \to \frac{1}{2}} \quad \text{for } i = 0.$$

To illustrate the fixed interval pdf discretisation, Fig. 3.19 provides an example. The function's domain is partitioned into several intervals each with a width d. The discretisation associates probabilities of the fixed intervals to multiples of d ($1d$, $2d$, $3d$, etc.). Each of these values gets the probability of the integral of the interval

$[(i-\frac{1}{2})d, (i+\frac{1}{2})d[$ around it. For example, the value $3d$ (i.e., $i = 3$) gets the probability of the interval's $[2.5d, 3.5d[$ integral, which is the striped area under the graph.

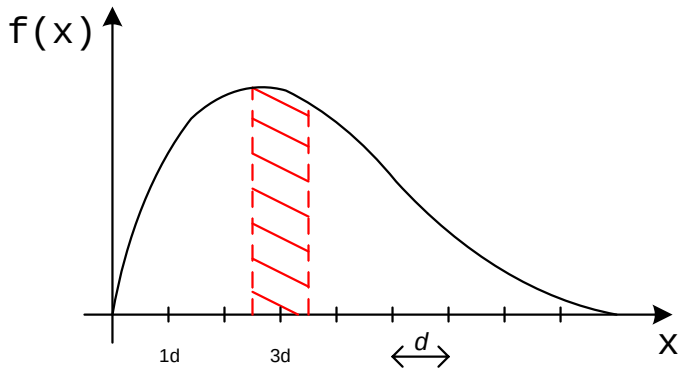


Figure 3.19: Example: pdf-discretisation with fixed intervals

`SamplePDFs` with fixed interval sizes are useful to implement arithmetic operations, which combine two pdfs. With the same distance d for each pdf, there are efficient convolution algorithms (Chapter 3.3.6). `SamplePDFs` are only used internally by analysis tools and are not specified directly by developers when modelling a system. Therefore, the StoEx-framework does not offer a concrete textual syntax for `SamplePDFs`. Developers can only specify `BoxedPDFs`. However, tools visualising the results of response time predictions always use `SamplePDFs` as basis for the result, because they are the output of the analysis tools.

Boxed Probability Density Function `BoxedPDFs` divide a pdf's domains into a set of intervals with variable sizes. Variable interval sizes sometimes offer a better pdf approximation using fewer values than fixed interval sizes. For example, pdfs with wide, almost constant parts on one hand and a few sharp peaks on the other hand would require a large number of fixed intervals for the constant parts using `SamplePDFs`. Instead, using `BoxedPDFs`, these large constant parts could be combined into a single large interval, while the sharp peaks could be easily approximated in detail with a number of few small intervals. This is especially convenient for developers entering pdfs into tools, because they have to specify a much smaller number of intervals without losing accuracy.

A `BoxedPDF` partitions a pdf's domain into a set I of $N \in \mathbb{N}$ non-overlapping intervals, so that there are no gaps between these intervals. That is

- $J_i \cap J_j = \varnothing \quad \forall J_i, J_j \in I; \quad J_i \neq J_j; \quad i, j \in \{1, ..., N\}$ (non-overlapping), and
- $\bigcup_{J \in I} = [0, x[\quad$ for $x \in \mathbb{R}^+ \quad$ (no gaps).

To assert these two properties, a `BoxedPDF` specifies each interval only with its right hand limit assuming that the left hand limit is the former interval's right hand limit or zero. This results in a list I_X with an ordered set of x_i, so that $x_1 < x_2 < ... < x_{N-1} < x_N$. Then the i-th interval is $[x_{i-1}, x_i[$ for $i > 1$ and $[0, x_i[$ for $i = 1$. The probability p_i for the i-th interval is

$$p_i = \int_{x_{i-1}}^{b} f(x)dx, \quad lim_{b \to x_i}, \quad \text{for } i > 1, \quad \text{and}$$

$$p_i = \int_{0}^{b} f(x)dx, \quad lim_{b \to x_1}, \quad \text{for } i = 1.$$

Fig. 3.20 contains an example of a `BoxedPDF`. It shows the concrete textual syntax for `BoxedPDFs` on top and below the corresponding graphical visualisation. The textual syntax uses `DoublePDF` as a keyword (the sample space is always $\mathbb{R}$, i.e., `double` for continuous random variables), and includes a set of pairs with the right hand limit of each interval and its probability enclosed in parenthesis. The specified `BoxedPDF` has discretisised the given pdf with five intervals starting from zero to a maximum value of $x = 93$. The wide, almost constant area is approximated by a single box, where a `SamplePDF` would have specified a large number of intervals.

3.3.6 Stochastic Expressions

The former subsections dealt with single random variables to model different uncertain properties of component-based systems. Besides single random variables, it is often desirable to specify a random variable as a combination of several other random variables using arithmetic or boolean operations. Especially for expressing dependencies between multiple properties (e.g., a parameter value influencing a resource demand), developers need to specify mathematical operations on random variables.

For example, a developer should be able to multiply a timing value giving as a random variable for a certain service execution by a factor, to model that different (faster or slower) hardware executes the service, so that it is possible to answer

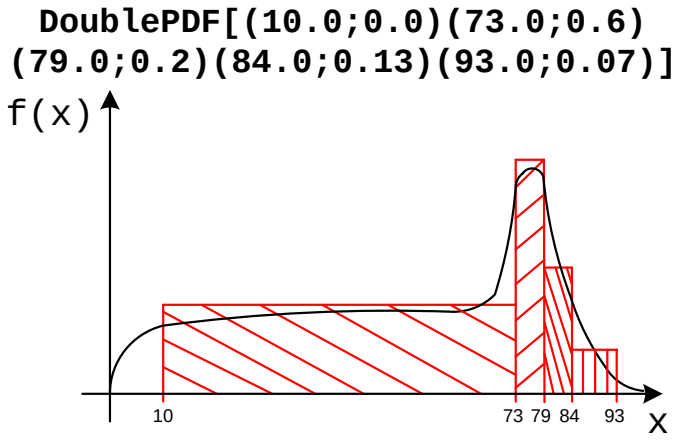


Figure 3.20: Example: pdf-discretisation with variable intervals

hardware sizing questions. As another example, RDSEFFs (Chapter 4.3) use boolean guards on control flow branches of services, which reference parameter value characterisations as random variables (Chapter 4.1). Therefore, it is necessary to compare random variables to model boolean guards.

Therefore, the StoEx-framework [RBH+07] includes a meta-model with a textual, concrete syntax to combine multiple random variables with different operations (e.g., addition, multiplication) to define new random variables. Fig. 3.21 depicts the meta-model and thus illustrates the abstract syntax of the so-called "Stochastic Expression Language" implemented by the StoEx-framework. `RandomVariables` contain their specification both as a string (attribute `specification`) and as a derived attribute, which yields an instance of the `Expression` meta-class possibly being a large object tree realising the abstract syntax tree of the stochastic expression. Developers enter the string representation into tools, from which the lexer and parser of the StoEx-framework create the object representation used for computations or model-transformations.

An `Expression` can be an `Atom` (cf. Fig. 3.21 bottom), or a complexer, composed term (cf. middle part) involving different mathematical operations. `Atoms` are literals representing constant random variables (`IntLiteral`, `DoubleLiteral`, `BoolLiteral`, `StringLiteral`, discrete or continuous random variables (`ProbabilityFunctionLiteral`), variable references (`Variable`, cf. Chapter 4.1), parenthesis to define precedences within term expressions (`Parenthesis`), or a predefined function (`FunctionLiteral`).

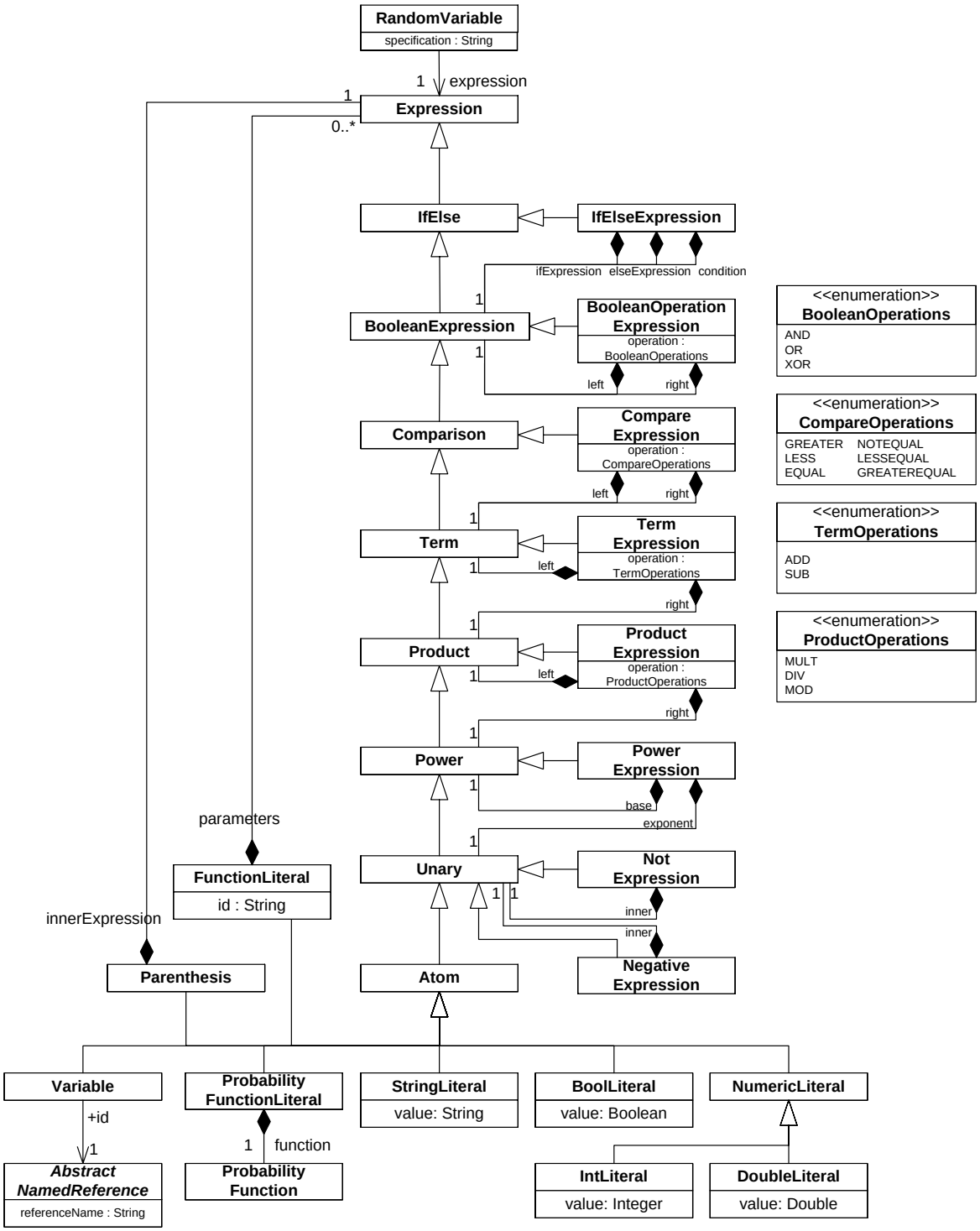


Figure 3.21: Random Variable and Stochastic Expressions Grammar (meta-model)

The underlying type system restricts the boolean, compare, and arithmetic operations to certain operands. For example, boolean operations are only allowed with boolean operands, while arithmetic operations are only allowed with numerical and probability function literals. The following describes the semantics for all operations allowed by the type system.

Boolean Operations Boolean operations ($\wedge, \vee,$ xor) are only valid for boolean random variables. Let A and B be two independent boolean random variables (i.e., $A : \mathbb{B} \to \mathbb{R}$, $B : \mathbb{B} \to \mathbb{R}$). In the following, $P(A)$ is a short-hand notation for $P(A = true)$. Let $C_\wedge, C_\vee, C_{\text{xor}}$ be boolean random variables defined as $C_\wedge = [A \wedge B]$, $C_\vee = [A \vee B]$, and $C_{\text{xor}} = [A \text{ xor } B]$. The probabilities of these events can be computed as

$$
\begin{aligned}
P(C_\wedge) &= P(A \wedge B) = P(A)P(B) \\
P(C_\vee) &= P(A \vee B) = P(A) + P(B) - P(A \wedge B) \\
P(C_{\text{xor}}) &= P(A \text{ xor } B) = P(A)(1 - P(B)) + (1 - P(A))P(B)
\end{aligned}
$$

In the StoEx-framework, these operations can be used for `ProbabilityMassFunctions` with a boolean domain (`BoolPMF`). A `BoolLiteral` can be expressed as boolean random variable D with either $P(D) = 1$ for $true$ or $P(D) = 0$ for $false$.

Compare Operations Compare operations ($>, <, =, \neq, \leqslant, \geqslant$) are valid for discrete or continuous random variables.

Discrete Random Variables: Let X and Y be two *independent* discrete random variables. Let A_1 be a boolean random variable defined as $A_= = [X = Y]$. The probabilities of this event can be computed as

$$
\begin{aligned}
P(A_=) &= \sum_{-\infty < x \leqslant \infty} P(X = x, Y = X) \\
&= \sum_{-\infty < x \leqslant \infty} P(X = x, Y = x) \\
&= \sum_{-\infty < x \leqslant \infty} P(X = x)P(Y = x).
\end{aligned}
$$

Analogously, the probabilities of the boolean random variables $A_\neq = [X \neq Y]$, $A_< = [X < Y]$, $A_\leqslant = [X \leqslant Y]$, $A_> = [X > Y]$, $A_\geqslant = [X \geqslant Y]$ are given by

$$
\begin{aligned}
P(A_{\neq}) &= \sum_{-\infty<x\leqslant\infty} P(X=x)P(Y\neq x)\\
P(A_{<}) &= \sum_{-\infty<x\leqslant\infty} P(X=x)P(Y> x)\\
P(A_{\leqslant}) &= \sum_{-\infty<x\leqslant\infty} P(X=x)P(Y\geqslant x)\\
P(A_{>}) &= \sum_{-\infty<x\leqslant\infty} P(X=x)P(Y< x)\\
P(A_{\geqslant}) &= \sum_{-\infty<x\leqslant\infty} P(X=x)P(Y\leqslant x)
\end{aligned}
$$

Notice, that these comparison operations are also valid for constant random variables. Therefore they define comparisons between constants and discrete random variables.

Arithmetic (Term and Product) Operations Arithmetic operations $(+, -, *, /, \%)$ are defined for discrete and continuous random variables.

Discrete Random Variables [Tri01]: Let X and Y be *independent*, discrete random variables. Let Z_+ be a discrete random variable defined as $Z_+ = X + Y$. The probability of the event $X + Y = t$ can be computed as

$$
\begin{aligned}
P(Z_+ = t) &= \sum_{-\infty<x\leqslant t} P(X=x, X+Y=t)\\
&= \sum_{-\infty<x\leqslant t} P(X=x, Y=t-x)\\
&= \sum_{-\infty<x\leqslant t} P(X=x)P(Y=t-x).
\end{aligned}
$$

Thus, the pmf for the sum of X and Y is given by

$$
p_{Z_+}(t) = p_{X+Y}(t) \;\; = \sum_{-\infty<x\leqslant t} p_X(x)p_Y(t-x).
$$

The included sum is also called discrete convolution. Analogously, the pmfs for $Z_- = X - Y$, $Z_* = X * Y$, $Z_/ = X/Y$ can be computed:

$$
\begin{aligned}
p_{Z_-}(t) = p_{X-Y}(t) &= \sum_{-\infty<x\leqslant t} p_X(x)p_Y(x-t)\\
p_{Z_*}(t) = p_{X*Y}(t) &= \sum_{-\infty<x\leqslant t} p_X(x)p_Y(\frac{t}{x}), \qquad x\neq 0, p_X(z)=p_Z(z)=0 \text{ for all } z\notin\Omega\\
p_{Z_/}(t) = p_{\frac{X}{Y}}(t) &= \sum_{-\infty<x\leqslant t} p_X(x)p_Y(\frac{x}{t}), \qquad t\neq 0, p_X(z)=p_Z(z)=0 \text{ for all } z\notin\Omega
\end{aligned}
$$

These computations are also valid for constant random variables (e.g., $Y(\omega) = c$, for all $\omega \in \Omega$). Notice, that some discrete random variables used in the PCM must not become negative (e.g., random variables for loop iteration numbers). Analysis methods need to check before computing the difference between two random variables X and Y, whether the largest i with $p_Y(i) > 0$ is smaller than the smallest j with $p_X(j) > 0$, so that the subtraction does not result in negative values. If the division of two random variables results in sample points outside the domain $\mathbb{N}$, the corresponding values have to be rounded to yield a valid new random variable.

Continuous Random Variables [Tri01]: Let X and Y be two *independent*, continuous random variables with the densities f_X and f_Y. Let Z_+ be a continuous random variable defined as $Z_+ = X + Y$ and A be a subset of $\mathbb{R}^2$ given by $A = \{(x, y)|x + y \leqslant z\}$. The distribution function of the Z_+ can be computed as

$$\begin{aligned}
P(Z_+ \leqslant z) &= P_{(X+Y)}(A) \\
&= \iint_A f_X(x) f_Y(y)\, dx\, dy && \text{(because of the independence)} \\
&= \int_{-\infty}^{z-x} \int_{-\infty}^{\infty} f_X(x) f_Y(y)\, dx\, dy && \text{(definition of } A\text{)} \\
&= \int_{-\infty}^{z} \int_{-\infty}^{\infty} f_X(x) f_Y(t-x)\, dx\, dt && (y = t - x) \\
&= \int_{-\infty}^{z} f_{Z_1}(t)\, dt
\end{aligned}$$

Thus, the density of $Z_+ = X + Y$ is given by

$$f_{Z_+}(z) = \int_{-\infty}^{\infty} f_X(x) f_Y(z-x)\, dx, \qquad -\infty < z < \infty.$$

This integral is also called the convolution of f_X and f_Y. Analogously, the densities for $Z_- = X - Y$, $Z_* = X * Y$, $Z_/ = X/Y$ can be computed:

$$\begin{aligned}
f_{Z_-}(z) &= \int_{-\infty}^{\infty} f_X(x) f_Y(x-z)\, dx, && -\infty < z < \infty \\
f_{Z_*}(z) &= \int_{-\infty}^{\infty} f_X(x) f_Y(\frac{z}{x})\, dx, && -\infty < z < \infty,\ x \neq 0 \\
f_{Z_/}(z) &= \int_{-\infty}^{\infty} f_X(x) f_Y(\frac{x}{z})\, dx, && -\infty < z < \infty,\ z \neq 0.
\end{aligned}$$

Power and modulo operations are undefined for probability distributions.

3.4 Summary

This chapter has set the context for the PCM extensions introduced in the next chapter. It described the targeted process model of the PCM, which proposes a division of the modelling task for a component-based software architecture to different developer roles. The process model is based on the process model by Cheesman et al. and adds a QoS analysis workflow. The PCM is aligned with this process model, i.e., it is divided into several domain-specific modelling languages for its developer roles. The meta-model allows describing interfaces and different types of software components as well as hardware resources, which are important for performance analysis. The PCM includes a special context model, which stores information for each component, which is necessary for performance predictions but cannot be specified by the component developer, such as the composition to other components, the allocated resources, and its usage. Finally, the behavioral description languages described in the next chapter rely on stochastic characterisation of user and component behaviour. Therefore this chapter has explained the basics of random variables, including discrete and continuous distribution functions. The StoEx framework provides means to combine individual random variables with boolean, compare, and arithmetic operators, which is necessary to specify the parameter dependencies described in the next chapter.

Chapter 4

Behavioural Models for Users and Components in CBSE

This chapter introduces extensions to the formerly described Palladio Component Model, which enable developers to model user and component behaviour. The extension particularly focusses on making the influence of the usage profile (i.e., the number of requests and the included parameter values) on the performance of component-based system explicit.

Chapter 4.1 motivates the need for a parameter characterisation model that abstractly models the performance-relevant properties of service parameters. The proposed extension for the PCM is used by both of the following modelling languages. Chapter 4.2 introduces the PCM usage modelling language for describing user behaviour. Chapter 4.3 presents the Resource Demanding Service Effect Specification (RDSEFF) modelling language for describing performance-relevant component behaviour. While these chapters describe the semantics of the languages with natural language, Chapter 4.4 defines their semantics formally with a mapping to queueing Petri nets.

4.1 Parameter Abstractions

Parameter abstractions are an important part of the user and component behavioural specification languages, which will be introduced in Chapter 4.2-4.3. Before using them in these languages, this section will first motivate their need and then provide an overview of modelling them in the PCM.

4.1.1 Motivation

Signatures of component interfaces specify formal parameters of component services. Furthermore, a component may include global (i.e., component-wide) variables, which may further parameterise all services of the component. There are three categories of parameters:

- **Input Parameters:** Clients (i.e., users or other components) pass input parameters when calling a component's service. This service uses these parameters to carry out its computations. In a service signature, these parameters can have an IN or INOUT modifier (Chapter 3.2.2).
- **Output Parameters:** Each component service has one return value and may additionally have several parameters with OUT or INOUT modifiers in its signature (Chapter 3.2.2). A component service returns values of these parameters to the caller, where they might influence further component behaviour.
- **Global Parameters:** Each component can have global, internal parameters, which are not declared in its interfaces, because they should stay hidden from clients. Each service implemented by the component may use them. Therefore, they additionally parameterise services.

For each formal parameter declared in a signature, component clients provide actual values instantiating the declared data types when calling the service. These actual parameter values can influence the service's QoS properties significantly. In particular, they can alter the following four QoS-relevant parts of component behaviour (Fig. 4.1):

- **Resource Demands:**[1] A service can alter its resource demands depending on parameters, which alters response times. A resource is a hardware device such as a CPU or hard disk. For example, the time for the execution of a service that allows uploading files to a server depends on the size of the files that are passed as input parameters (i.e., resource demand $\approx x + \frac{\text{size(inputFile)}}{\text{HD-throughput}'}$ where x is some constant overhead per call). In this case, the parameter alters the demand issued to the storage device.

[1] In an RDSEFF, component developers specify resource demands instead of timing values to keep the specification context-independent (i.e., not bound to a specific resource environment). Refer to Chapter 4.3 for details.

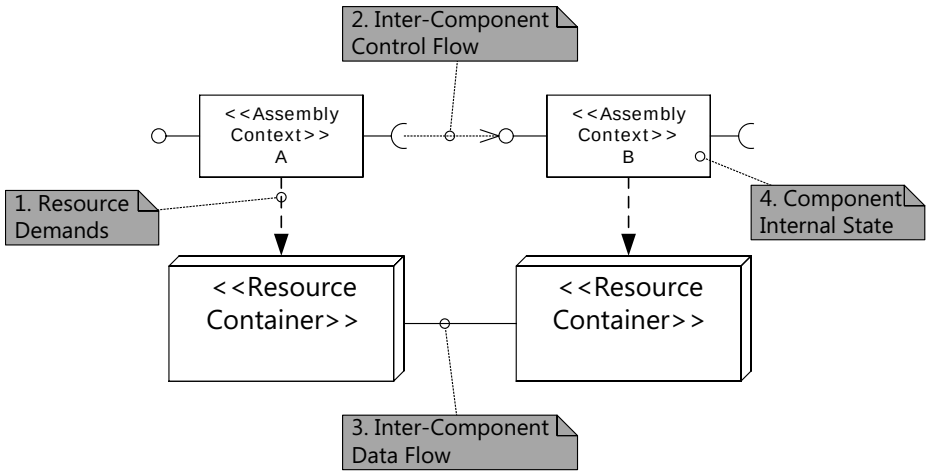


Figure 4.1: Influences on QoS-relevant Component Behaviour by Parameters

- **Inter-Component Control Flow:** A component service pass the control flow to different other components depending on parameter values. For example, a component service might provide clients access to a number of databases, thus communicating with several database interfaces as required services. This service could call different required services depending on the input parameter passed to it (e.g. requests for large files get directed to DB1 while requests for small files get directed to DB2). Another example could be a component service having a collection parameter, which would call another component's service subsequently for each item in the array. In this case, an input parameter changes inter-component control flow. Notice, that int*ra*-component control flow should be abstracted in a behavioural component specification to retain the black-box principle.

- **Inter-Component Data Flow:** A service can pass different parameters to its required services depending on its own parameters. Furthermore, a service can receive different parameters from a required service depending on passed input parameters. If two interacting components are deployed on different servers, these parameters have to be transferred via network connections, which has an impact on performance. For example, a software architect can arrange multiple components in a pipe-and-filter pattern, so that the output of one component is the input of the next component. If large data packets are transferred between these components over network connections with limited bandwidth, the perceived responsiveness of the whole system will decrease.

- **Component Internal State:** A service can store the value of a received parameter as part of the component's internal state. Consider a component allowing users to log into a system, which stores user sessions as global variables. The later behaviour of other services of this component in terms of control flow propagation and resource demand could depend on which user is currently logged in. Thus the QoS properties of the component are related to the internal parameter, which was created when the user logged into the system.

Because of these influences, the behavioural dependencies on parameters should be included into component QoS specifications. However, as elaborated in Chapter 2.5, many existing performance prediction approaches and component specification languages disregard the influence of parameter values.

In CBSE, component developers can and should not know how clients use their components. When they create performance specifications of their components, they need to include explicit dependencies on parameters. Domain experts can then model actual values supplied by users of a formal parameter. With this information, tools can solve the dependencies in the specifications. With explicit parametric dependencies, the component performance specifications produced by component developers remain context-independent (i.e., from a specific usage) and reusable.

Notice that for a performance prediction model, domain experts do not have to model an actual parameter for each formal parameter, but only for parameters significantly influencing performance properties. Many parameters do not influence performance properties and therefore component developers do not have to include dependencies to them into their component performance specifications. Component developers have to assess, which parameters are performance-relevant. With the reduced number of parameters, the performance prediction model remains analysable. By neglecting performance-irrelevant parameters, it does not lose accuracy. As a side-effect, the component performance specifications stays abstract and preserves the component developer's intellectual properties.

4.1.2 Parameter Characterisation

There are several requirements to create expressive parameter characterisations for performance prediction:

- **Value and Meta-Data:** Domain experts can directly characterise parameter values (e.g., `anInt.VALUE = 1` or `aString.VALUE = 'foo'`) or provide

descriptive meta-data about a parameter. Especially for performance prediction, a parameter's value is often not suitable for analysis. For example, if a component writes a file to a disk, its size in bytes is needed (e.g., `aFile.BYTESIZE = 100`) instead of its value to calculate the execution time of the disk. As another example, a component developer needs to specify the resource demand of a component service iterating over a collection in dependency to the number of elements contained in the collection (e.g., `aCollection.NUMBER_OF_ELEMENTS = 5`). In such cases, the actual value of the parameter is not useful for the performance model. For collection or composite parameters, it is in general impractical to specify their actual value instead of meta-data.

- **Random Variables:** Besides using constants to characterise parameter values or meta-data, it is often more expressive to use discrete random variables and their probability distributions. For example,

```
aNumber.VALUE = IntPMF[(1;0.2)(2;0.5)(3;0.3)]
```

characterises the value of parameter `aNumber` with a pmf, meaning that users instantiate `aNumber` with "1" with a 20 percent probability, "2" with a 50 percent probability, and so on (Chapter 3.3). Instead of integers, a domain expert can use other data types, for example enums:

```
aName.VALUE = EnumPMF[('A';0.2)('B';0.5)('C';0.3)]
```

Furthermore, random variables are also useful to characterise meta-data, for example:

```
aCollection.NUMBER_OF_ELEMENTS = IntPMF[(10;0.3)(20;0.7)]
```

Using random variables makes parameter modelling more expressive and covers the uncertainty of domain experts when describing the usage of a system during early development stages. It is furthermore a convenient way to specify parameter characterisation variations for larger user groups, so that not a single constant parameter characterisation needs to be provided for each user.

- **Subdomain grouping:** The domain of a parameter value or meta-data is usually very large. For example, integer values may range over $2^{64} - 1$ values, strings are usually only limited by the available memory. As in black box testing, it is necessary for performance models to partition large domains into a manageable number of subdomains. For example,

```
aNumber.VALUE = IntPMF[([INT_MIN-0;0.2)([1-INT_MAX;0.8)]
```

  partitions the integer domain into two subdomains $[-2^{31}, 0]$ and $[1, 2^{31} - 1]$. The same is possible for strings:

```
aName.VALUE = EnumPMF[('Starts with A';0.2)('Starts with B';0.5)('Starts with C';0.3)]
```

  partitions the string domain into four subdomains: three for starting strings with different letters, and a forth, implicit subdomain for all other strings with a probability of zero. Subdomains or ranges can also be defined for meta-data. For performance prediction, parameter domains should be partitioned in such a way, that parameter values of a single partition lead to similar performance behaviour (also see Chapter 4.1.4).

- **Extensible Meta-Data:** The kinds of meta-data specifications need to be fixed in any component performance meta-model, so that different component developers can refer to meta-data unambiguously and tools may automatically evaluate the specifications. However, besides `BYTESIZE` and `NUMBER_OF_ELEMENTS`, there is further (and potentially unlimited more) possible meta-data about parameters, which may influence the performance of a software component. For example, if a component service implements a quicksort algorithm, its resource demand depends on whether the collection to be sorted is already presorted or not. Such a structural property needs to be expressed as an additional meta-data attribute. Therefore, any performance specification language should provide a mechanism to create new parameter characterisation types and store them globally in repositories, where different component developers can refer to them.

4.1.3 Parameter Characterisation in the PCM

Variable Usage The PCM tries to realise the four requirements described above. In the PCM, a `VariableUsage` realises a parameter characterisation (Fig. 4.2). It consists of a parameter name (`AbstractNamedReference`) and a number of characterisations (`VariableCharacterisation`).

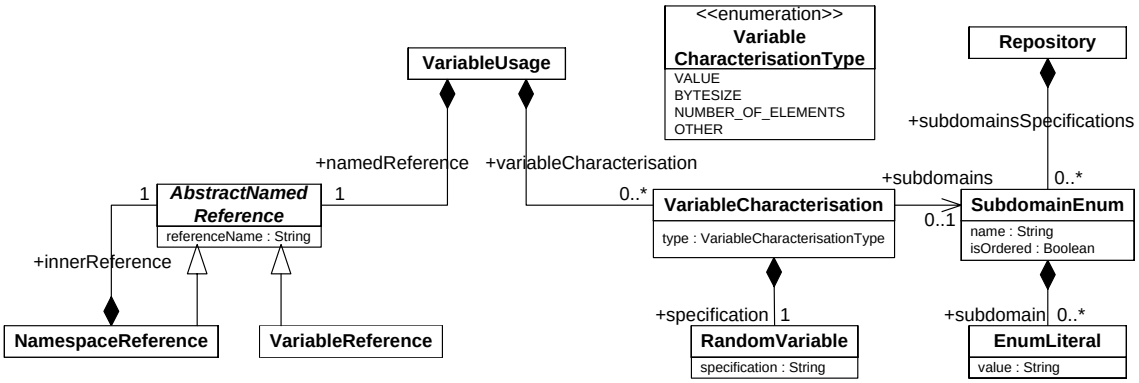
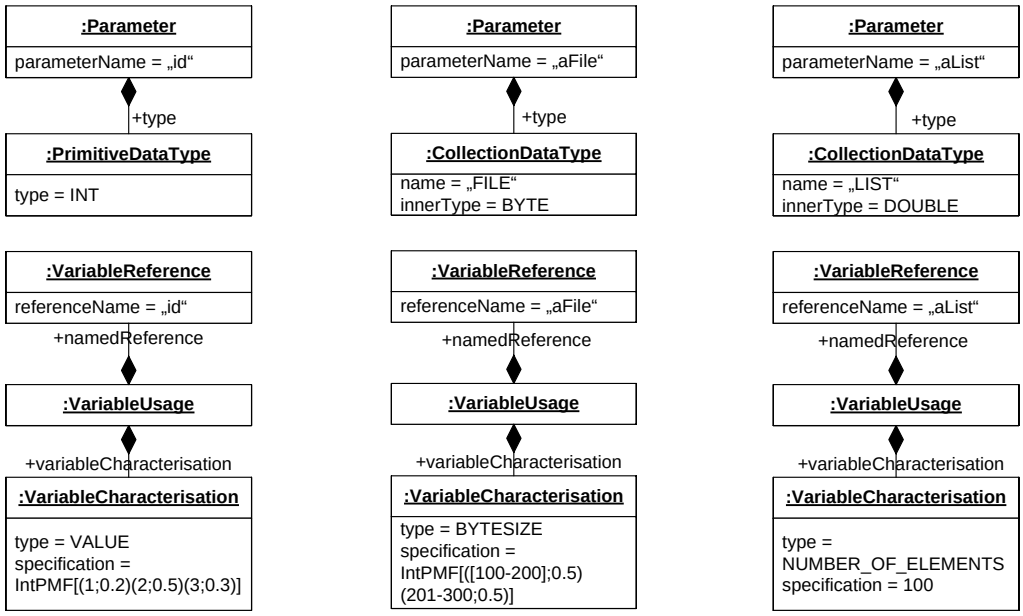


Figure 4.2: Variable Usage (meta-model)

Parameter names (e.g., aNumber) are strings (attribute `referenceName` of `VariableReference`). These names may include a namespace declaration (e.g., namespace1.namespace2.aNumber), which is needed because of the possible data type nesting within collection and composite data types. For example, a domain expert can provide a `VariableUsage` for each inner declaration of a composite data type. In this case, the composite data type's name is the `NamespaceReference`, and the inner declaration's names are the `VariableReferences`. For example, in the variable name x.y.z, x and y are `NamespaceReferences` (representing composite data types), while z is a `VariableReference`.

A `VariableCharacterisation` contains a `RandomVariable`'s specification. Parameter characterisation types are included as so-called `VariableCharacterisationTypes`. The PCM directly supports `VALUE`, `BYTESIZE`, and `NUMBER_OF_ELEMENTS`, because these are common characterisation types. Example instances of different parameter characterisations are given in Fig. 4.3.

Additionally, the PCM provides a mechanism to include further parameter characterisations. `VariableCharacterisations` with the type `OTHER` (and only these) reference a so-called `SubdomainEnum` (Fig. 4.2). Its name is the keyword for a new parameter attribute (e.g., "MP3-BITRATE"). It contains a number of `EnumLiterals` partitioning this parameter attribute's domain into a finite number of



(a) VALUE (b) BYTESIZE (c) ELEMENTS

Figure 4.3: Examples for Variable Characterisations

subdomains (e.g., ”128Kbps”, ”192Kbps”, ”256Kbps”, ”other”). Component developers need to include such `SubdomainEnums` into PCM `Repositories`, so that different component developers can use them by referencing the corresponding `Repository` in their specifications.

The subdomains introduced by a `SubdomainEnum` should represent element sets of the parameter characterisation's domain for which a similar behaviour in terms of performance is expected. This partitioning must span the whole domain. Therefore, including an ”other” subdomain, subsuming all elements not included in a specific subdomain, into such a partitioning is often useful. It is further possible to specify whether the set of subdomains is ordered. An ordered set of subdomains enables subsuming subdomains to simplify specifications. For example, a component developer could specify that a service behaves differently for MP3 files with a bitrate of smaller than ”256Kbps”, thereby combining the subdomains of ”128Kbps” and ”192Kbps”.

With a subdomain specification, domain experts can characterise parameters (e.g., with ”MP3-BITRATE”) by providing the specification of a discrete random variable `EnumPMF`, where the random variable's range is the set of specified subdomains. For example, a domain expert could specify the bitrate of users' MP3 files as:

```
aFile.MP3-BITRATE = EnumPMF[(''128Kbps'';0.3)(''192Kbps'';0.3)(''256Kbps'';0.3)(''other'';0.1)]
```

The mechanism for introducing new parameter characterisations is flexible, so that component developers can include different performance influencing factors of parameters into their specifications. The PCM's design cannot foresee all possible parameter characterisations in advance.

An example for a parameter characterisation specifically for collections would be a `SubdomainEnum` called "SORTING" with the subdomains "sorted" and "unsorted". It could be used for accurately specifying the performance of component sorting services, which alter their behaviour depending on the sorting degree of collections passed to them as parameters (e.g., quicksort). A domain expert could specify whether collections supplied by component clients are sorted or not, thereby refining the performance specification and enabling more accurate prediction results.

Another parameter characterisation could be the parameter's data type, in case a subtype is passed as a parameter to a service, which expects a supernate as parameter. For example, a service drawing arbitrary graphical objects changes its performance depending on the different kinds of graphical objects passed to it. It can draw rectangles faster than circles. To make an accurate performance specification, the set of possible subtypes needs to be finite and explicitly specified. A component developer could introduce a `SubdomainEnum` named "GRAPHICSTYPE" with the subdomains "circle", "rectangle", "polygon", etc. In the behavioural performance specification of a component service, the developer could refer to these subdomains and specify a different resource usage for each of them. The domain expert could provide different probabilities for each kind of graphical object.

Uses of Variable Usage in the PCM Developers can attach `VariableUsages` to different elements in PCM instances. Not only domain experts, but also software architects and component developers may model `VariableUsages` in their respective domain specific languages. The following provides an overview on how parameter characterisations are included in the PCM (Fig. 4.4).

In a component-based system, *input parameters* may either be supplied by users or other systems when invoking services at the system boundaries or by component services when calling required services. Therefore, in the PCM, `EntryLevelSystemCalls` from the `UsageModel` (Chapter 4.2) and `ExternalCallActions` from the `RDSEFF` (Chapter 4.3) contain `VariableUsages`.

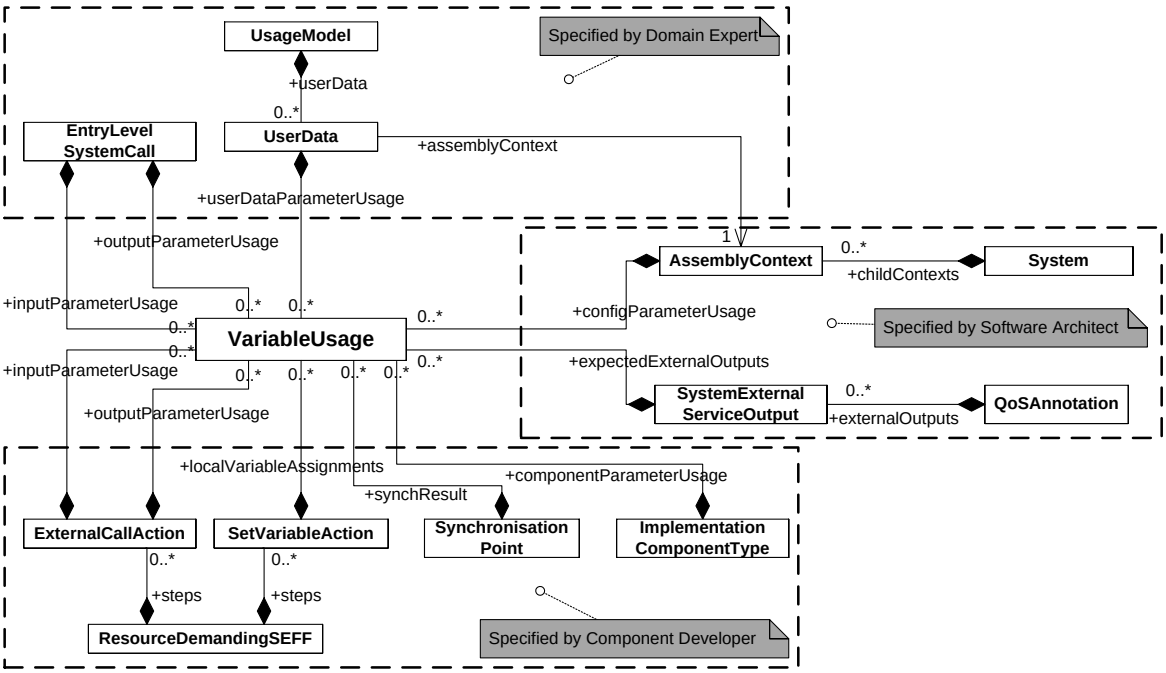


Figure 4.4: Classes containing Variable Usage (meta-model)

The PCM allows to attach `VariableUsages` characterising *output parameters* at various locations: In the `UsageModel`, `EntryLevelSystemCalls` can contain output parameter characterisations coming from the system. In `RDSEFFs`, `ExternalCallActions`, `SetVariableAction`, and `SynchronisationPoints`: can contain output `VariableUsages`. Chapter 4.3 will explain them in detail. Furthermore, a system may receive output parameter values from other systems. The software architect can characterise these values with a `QoSAnnotation` and its contained `SystemExternalServiceOutputs`.

Component Parameters For *global parameters*, the PCM provides a special way of modelling, which is explained in the following. The internal state of a component can influence its performance significantly. For performance models, the elements of component internal state can be perceived as an additional input parameters to each of the component's services. Component developers could specify dependencies between internal state and resource demands or component behaviour.

Modelling component internal state accurately can easily complicate performance models so much that they become intractable: A component may hold an internal state for each user requesting services. In web-based systems, this is often

referred to as a user session. As performance models often aim at analysing systems with a high number of concurrent users, storing the internal state of all components for all users quickly becomes problematic. For example, in a component-based system with 10 components and 1000 concurrent users, an analysis tool would have to store 10000 internal state entities.

Additionally, the internal state of each component may consist of a large number of variables. For example, the contents of a whole database can be perceived as the internal state of a database component. Consider only 10 variables for each internal state, and above example expands to 100000 variables to store.

Furthermore, each service invocation may potentially alter the variables' values (i.e., each of the 100000 values). The state space of a component can be very large. Therefore, modelling internal state and state changes accurately for industrial size systems is impractical, because the resulting models quickly become analytically intractable.

The PCM does not support modelling components at runtime, when they hold a specific internal state. It aims at design-time performance predictions, and abstracts from specifics of component runtime instances. At the lowest abstraction level, PCM `Systems` model components at deployment time, i.e., there may be multiple deployment (but not runtime) instances of a single component type, which are each encapsulated by an `AssemblyContext` (Chapter 3.2.4).

To include a notion of internal state for a performance model, this work introduces static component parameter abstractions. Developers can attach such abstractions to any component (in an `AssemblyContext`) and extend the input domains of each of the component's services.

To create an abstraction from runtime state and keep the performance model tractable, the specification of these component parameter characterisation's random variables are equal for all users. Equal component parameter characterisations for large user groups are a strong abstraction from reality. However, in a single use case, which is usually the boundary of a performance prediction, it is sometimes sufficient to approximate the contents of a database as equal for all users. It is possible to model static component parameter abstractions with random variables, so that developers can express a certain degree of uncertainty.

Additionally, services cannot change static component parameter abstractions as they are assumed to stay constant for a specific performance analysis scenario (e.g., a use case). Unchangeable component parameter characterisations (i.e., no state changes, fixed random variable) in a scenario are another strong abstraction from

reality. It might not hold for transactional scenarios, where different component states in a single use case can influence perceived performance characteristics heavily. However, in some scenarios it is sufficient to express internal state as constant for the purpose of performance prediction.

The PCM supports specifying component parameters at the following locations, which mainly differ in the developer role, who is responsible for the specification (Fig. 4.4):

- `ImplementationComponentType`: A component developer can attach a number of `VariableUsages` to `ImplementationComponentTypes`. On the one hand, these parameter characterisations publish the names of all possible component parameters with their `AbstractNamedReferences`. On the other hand, they contain `VariableCharacterisations`, which model default characterisations (i.e., random variables) of the component parameter. These default characterisations are valid for each `AssemblyContext`, which references the `ImplementationComponentType`.

  For composite components, the `VariableUsages` of the inner `AssemblyContexts` are not visible from the outside, because it is implementation specific, that the component is composed from inner components. However, component developers can use the `VariableUsages` of `CompositeComponents` to delegate to inner `VariableUsages`. For example, composite component A contains an inner component B, which publishes a component parameter named $B.X$. The developer of the composite component can attach a component parameter $A.Y$ to the composite component and characterise the inner component parameter with `B.X.VALUE = A.Y.VALUE`. This is possible, because $B.X$ is visible to the component developer, who assembles the composite component, but not to the software architect using the component. The software architect using the composite component can then provide a characterisation for `A.Y.VALUE`, which will be delegated to `B.X.VALUE`.

- `AssemblyContext`: A software architect can overwrite the component developer's default component parameter characterisations when building a `System`. Therefore, the software architect attaches a `VariableUsage` to an `AssemblyContext`. It uses the same `AbstractNamedReference` as the component developer used for the default characterisation, but provides a new `VariableCharacterisation` with a different random variable specification. Component parameter characterisations provided by software archi-

tects refer to technical concepts and may for example be configuration parameters of a component. These parameters characterisations are usually not probabilistic. For example, a software architect may overwrite a component parameter characterisation, which activates or deactivates the component's logging mechanism, which influences the component's performance.

- `UserData`: A domain expert can also overwrite the component developer's default component parameter characterisations. Other than the software architect, the component parameter characterisations used by the domain expert refer to non-technical concepts and may for example model user data (i.e., global component data, which refers to the business aspects of the system). Domain experts model these component parameter abstractions in the `UsageModel`, which contains a set of `UserData`. Each `UserData` entity references an `AssemblyContext` and contains a `VariableUsage`. As for the software architect, this `VariableUsage` uses an `AbstractNamedReference` equal to one specified by the component developer for the component included in the `AssemblyContext`. The `VariableCharacterisation` of this `VariableUsage` overwrites the component developer's default characterisation. For example, a domain expert could use such a non-technical component parameter to characterise the number of customers. This parameter could be related to the size of a database table, so that the execution time for searching the table would change in dependency to the parameter.

Inner Characterisations The PCM provides the key word `INNER` for parameters with a collection data type. It allows an additional characterisation of the inner elements of a collection, besides characterising the collection itself. For example, the following characterisation specifies the byte size distribution of all inner elements in a collection:

```
aCollection.INNER.BYTESIZE = IntPMF[([0-9];0.3)([[10-19];0.3)([20-30];0.4)]
```

It means that each contained file's byte size is distributed between 0 and 30 bytes with the given probabilities. The inner element is a representative for all elements within a collection. It is included for convenience reasons, so that developers may characterise inner elements without providing a single characterisation for each inner element.

Characterisations of `INNER` elements in collections lead to stochastic dependencies between two occurrences of this characterisation. For example, if a component

developer uses an INNER characterisation on two successive occasions in a control flow when specifying component performance, the characterisation of the second occurrence is stochastically dependent on the characterisation of the first occurrence, if none of the calls manipulates the characterisation.

Due to the mathematical complexity (i.e., state space explosion) when evaluating stochastically dependent variables, the tools solving the specifications INNER always assume stochastic independence between successive uses of INNER. The only exception are CollectionIteratorActions (Chapter 4.3), for whose body behaviour stochastic dependence is assumed.

4.1.4 Implications of Parameter Abstractions

Different component developers have to agree across component boundaries, which parameter characterisation types they use in their specifications.

If a component A calls a service of component B, then the developer of A needs to know what parameter characterisations types were used in the specification of B. This is necessary to ensure that a system model is completely specified, and tools can resolve all parameter dependencies. Consider a service specification of B, where a resource demand is specified in dependency to the BYTESIZE of parameter x, but component A has only specified a VALUE of x. The specification is incomplete, as tools cannot resolve the parameter dependency on the resource demand in B, because the BYTESIZE specification of x is missing from component A.

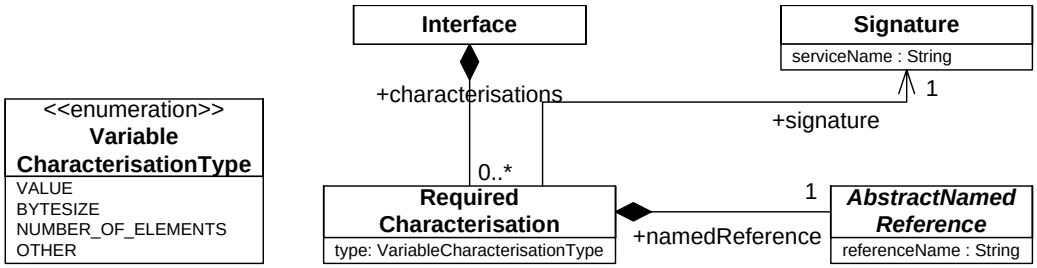


Figure 4.5: Required Characterisations (meta-model)

Therefore, component developers need to specify the expected characterisation types for each parameter of a service signature in the interface, because interfaces do not depend on components and are a contractual agreement among different component developers. In the PCM, developers can use Required-Characterisations (Fig. 4.5) in Interfaces to specify needed characterisation

both for developers implementing the component as well as for clients using the component. These characterisations contain an `AbstractNamedReference` (see also Fig. 4.2), which represents the name of the characterised parameter.

Besides specifying required parameter characterisations in interfaces, it is furthermore desirable to derive the needed subdomains for the parameter characterisations in the `UsageModel` from a complete `System` specification. This helps domain experts in specifying their variable characterisations, as they get hints for which parameter subdomains they have to provide probabilities. For example, if a component service S changes its behaviour depending on a input parameter X taking values below zero ($X < 0$) and larger than zero ($X \geqslant 0$), this construct defines two subdomains for which probabilities have to be provided.

It is therefore possible to implement algorithms traversing a fully specified system and tracing the needed subdomain specifications back to the usage model or component parameters. However, this has not been realised in this work and is subject to future work.

4.2 Usage Model

The usage of a software system by external clients has to be captured in models to enable model-driven performance predictions. Here, the term usage refers to workload (i.e., the number of users concurrently present in the system), usage scenarios (i.e., possible sequences of invoking services at system provided roles), waiting delays between service invocations, and values for parameters and component configurations. Chapter 2.5.3 has already discussed different approaches for modelling software system usage including operational profiles, Markov usage models, and UML.

This work introduces a new usage specification language, which (i) provides more expressiveness for characterising parameter instances than previous models, but (ii) at the same time is restricted to concepts familiar to domain experts to create a domain specific language. The language is called PCM usage model. Chapter 4.2.1 introduces its abstract syntax and explains its design rationale. Chapter 4.2.2 provides an example PCM usage model. Chapter 4.2.3 briefly discusses the relationship to similar modelling languages.

4.2.1 Meta-Model: Abstract Syntax and Informal Semantics

Fig. 4.6 shows the meta-model of the PCM usage model specified in Ecore. The usage model consists of a number of concurrently executed usage scenarios and a set of global user data specifications. Each usage scenario includes a workload and a scenario behaviour. The usage model specifies the whole user interaction with a system from a performance viewpoint. The following will explain each of the included concepts.

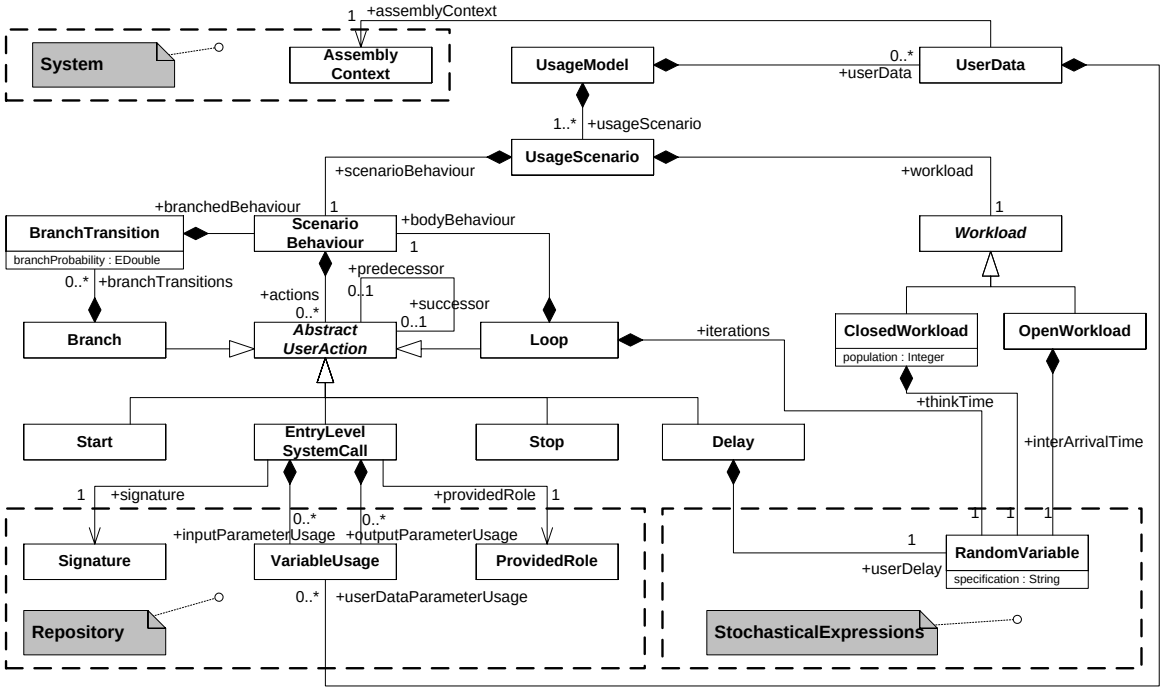


Figure 4.6: Usage Model (meta-model)

A `Workload` specifies the usage intensity of a system, which relates to the number of users concurrently present in the system. The PCM usage model adopts this concept from classical queueing theory [LZGS84]. The specified workloads can directly be used in queueing networks or easily be mapped to markings in stochastic Petri nets. Workloads can either be open or closed:

- `OpenWorkload`: Specifies usage intensity with an inter-arrival time (i.e., the time between two user arrivals at the system) as a `RandomVariable` with an arbitrary probability distribution. It models that an infinite stream of users

arrives at a system. The users execute their scenario, and then leave the system. The user population (i.e., the number of users concurrently present in a system) is not fixed in an `OpenWorkload`.

- `ClosedWorkload`: Specifies directly the (constant) user population and a think time. It models that a fixed number of users execute their scenario, then wait (or think) for the specified amount of think time as a `RandomVariable`, and then reenter the system executing their scenario again. Performance analysts use closed workloads to model scenarios, where the number of users is known (e.g., a fixed number of users in a company).

The algorithms used to analyse queueing networks differ depending on whether open or closed workloads are modelled [LZGS84]. Some special queueing networks can only be analysed given a particular workload type (open or closed). Notice, that it is possible to specify a usage model with open workload usage scenarios and closed workload usage scenarios at the same time. Open and closed workloads can be executed in parallel when analysing the model.

A `ScenarioBehaviour` specifies possible sequences of executing services provided by the system. It contains a set of `AbstractUserActions`, each referencing a predecessor and successor (except the first and last action), thereby forming a sequence of actions. Chapter 4.3.1 explains why it is advantageous to model control flow in this way, as the same principle is used in the RDSEFF language. Concrete user actions of the usage model are:

- `Branch`: Splits the user flow with a XOR-semantic: one of the included `BranchTransitions` is taken depending on the specified branch probabilities. Each `BranchTransition` contains a nested `ScenarioBehaviour`, which a user executes once this branch transition is chosen. After execution of the complete nested `ScenarioBehaviour`, the next action in the user flow after the `Branch` is its `successor` action.

- `Loop`: Models a repeated sequence of actions in the user flow. It contains a nested `ScenarioBehaviour` specifying the loop body, and a `RandomVariable` specifying the number of iterations.

- `EntryLevelSystemCall`: Models the call to a service provided by a system. Therefore, an `EntryLevelSystemCall` references a `ProvidedRole` of a PCM `System`, from which the called interface and the providing component

within the system can be derived, and a `Signature` specifying the called service. Notice, that the usage model does not permit the domain expert to model calls directly to components, but only to system roles. This decouples the `System` structure (i.e., the component-based software architecture model and its allocation) from the `UsageModel` and the software architect can change the `System` (e.g., include new components, remove existing components, or change their wiring or allocation) independently from the domain expert, if the system provided roles are not affected. `EntryLevelSystemCalls` may include a set of input parameter characterisations and a set of output parameter characterisations (as described in Chapter 4.1.3).

- `Delay`: Represents a timing delay as a `RandomVariable` between two user actions. The `Delay` is included into the usage model to express that users do not call system services in direct successions, but usually need some time to determine their next action. User delays are for example useful, if a performance analyst wants to determine the execution time for a complete scenario behaviour (instead of a single service), which needs to include user delays.

So far, `ScenarioBehaviours` do not include forks in the user flow (i.e., splitting the flow with an AND semantic), as it is assumed that users always act sequentially.

Besides `UsageScenarios`, the `UsageModel` includes a set of `UserData` to characterise data used in specific assembly contexts in the system. This data is the same for all `UsageScenarios`, i.e., multiple users accessing the same components access the same data. This `UserData` refers to component parameters of the system publicized by the software architect (Chapter 4.1.3). The domain expert characterises the values of component parameters related to business concepts (e.g., user specific data, data specific for a business domain), whereas the software architect characterises the values of component parameters related to technical concepts (e.g., size of caches, size of a thread pool, configuration data, etc.).

4.2.2 Example

To complete the description, Fig. 4.7 contains an example instance of the PCM usage model. The illustration uses a concrete syntax similar to UML activity diagrams, because the abstract syntax of this model instance is more complex and less intuitive. Each graphical element contains a stereotype (enclosed in angle brackets,

<<Metaclass>>) indicating the corresponding meta-class from the usage model. Keep in mind that there is no relation in this example to the UML meta-model, despite using a similar graphical representation.

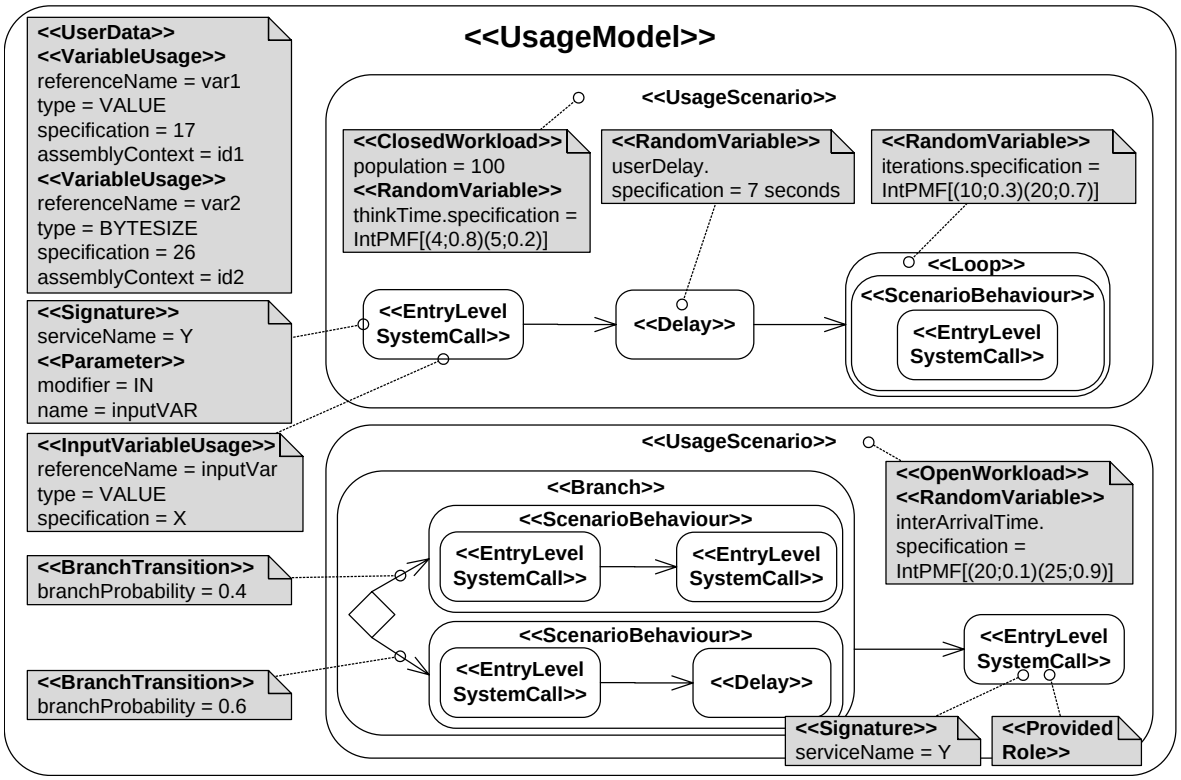


Figure 4.7: Usage Model (Example)

The example in Fig. 4.7 contains at least one representative instance for each meta-class from the usage model (compare with Fig. 4.6). It depicts a usage model with two usage scenarios and a user data specification containing two variable usages. A usage model may have an arbitrary number of usage scenarios. However, the analytical tractability of the model decreases when increasing the number of scenarios.

The domain experts has specified that the first usage scenario in the upper area of Fig. 4.7 executes as closed workload with 100 concurrent users (user population). Each user enters the scenario, executes all actions, and then waits for the think time specified as a `IntPMF`, before reentering the scenario again at the first action.

The first action of this scenario is a call to a service provided by the system. The call includes an characterisation for an input parameter of the service (`inputVar.VALUE = X`). Domain experts can use different characterisation types

as described in Chapter 4.1.3 and use probability distributions for the values as described in Chapter 3.3.

The second action of this scenario is a delay and specifies that each user waits exactly 7 seconds, before executing the next action. The user might for example read the screen output of the previous action for 7 seconds, before clicking a button that invokes the next action. The scenario concludes with a loop containing a call to a system service, which is invoked repetitively. According to the loop iterations specification, the user calls it either 10 times with a probability of 0.3 and 20 times with a probability of 0.7.

The second usage scenario in the lower area of Fig. 4.7 is executed as open workload. Users repeatedly arrive at the system and start invoking the scenario. The domain expert has specified an inter arrival time of 20 seconds with a probability of 0.1 and 25 seconds with a probability of 0.9. After executing the chain of actions in the scenario, each users exits the system and vanishes.

The first action of the second usage scenario is a branch. It contains two branch transitions each including a branch probability. It models a user choice with stochastic means, specifying that the users decides for the actions of the first transition with a probability of 0.4 or for the actions of the second transition with a probability of 0.6. The inner behaviours of both branch behaviours contain a call to a service provided by the system followed by a delay. Further, necessary annotations for these user actions have been omitted for clarity.

After executing one of the branched behaviours, each user executes another entry level system call in this scenario. When the system returns after executing the action, the user exits the system and does not return. Because of the unlimited number of arriving users in an open workload scenario, however, new users enter the system constantly.

4.2.3 Discussion

Notice, that unlike other behavioural description languages for performance prediction (e.g., [PW04, SLC+05, GMS05]), the PCM usage model specifically models *user* behaviour and for example does not refer to resources. Other performance meta-models mix up the specification of user behaviour, component behaviour, and resources, so that a single developer role (i.e., a performance analyst) needs to specify the performance model. Opposed to this, the PCM targets a division of work for multiple developer roles (cf. Chapter 3.1).

Furthermore, none of the other performance meta-models support explicit service parameter modelling. While CSM [PW04] includes a meta-class `Message` to specify the amount of data transferred between two steps in the performance model, and KLAPER [GMS05] allows the specification of parameter values in principle, none of these language uses the information to parameterise resource demands or component behaviour. Additionally, they do not provide the information readily analysable by MDSD tools.

4.3 Resource Demanding Service Effect Specification

This subsection introduces a new behavioural description language (RDSEFF) for component services, which is specifically designed for performance analysis. To predict the performance of a component-based system, it is useful to have such a modelling language for individual services instead of a monolithic model for the whole architecture, because it exploits benefits of CBSE, such as reusability, changeability, higher accuracy, and division of work (Chapter 2.4).

Component developers specify RDSEFFs for basic components and put them into repositories, where software architects can retrieve them and compose them into architectures (Chapter 3.1). RDSEFFs allow a parameterisation for the performance influencing factors (Chapter 2.4.2), which are outside the component developer's control. This factors include the performance of required services, the values of service parameters, and the performance of the deployment platform.

A special feature of RDSEFFs is the parameterisation for different usage profiles, which has been added as part of this thesis' contribution and allows a more expressive specification than other languages (cf. Chapter 2.5.1). RDSEFFs provide modelling elements to specify dependencies between characterisations of input (or component) parameters as part of the usage profile. Furthermore, they allow modelling resource demands, branch conditions, loop iterations numbers, and parameters passed to required services in dependency to parameter value characterisations. Thus, an RDSEFF can be easily adapted for different usage profiles, because it changes the modelled performance attributes (such as resource demands, branch probability, etc.) when automatically solving the specified dependencies (Chapter 6.2) with different input parameter characterisations.

The internal state of a component is a result of its usage, but it is not modelled in an RDSEFF. RDSEFFs describe the behaviour of an individual service, whereas internal state of a component may be referenced by different services of a component

and needs to be specified component-wide. The PCM uses component parameters (Chapter 4.1.3) to provide an abstraction from a component's internal state. Component developers, domain experts, and software architects can use them to specify parametric dependencies in RDSEFFs.

The following first describes the abstract syntax of the RDSEFF's meta-model and informally describes its semantics (Chapter 4.3.1). Afterwards, an example instance of the RDSEFF meta-model illustrates its modelling capabilities (Chapter 4.3.2). Chapter 4.3.3 compares the parameter dependency modelling of RDSEFFs to related work, before Chapter 4.3.4 lists several limitations of the language. Chapter 4.3.5 discusses the application of RDSEFFs in software engineering practice.

4.3.1 Meta-Model: Abstract Syntax and Informal Semantics

Fig. 4.8 shows the meta-model of the PCM RDSEFF specified in Ecore. Notice that some meta-classes (e.g., `VariableUsage`, `RandomVariable`) are duplicated in this diagram (but not in the model) to avoid visual clutter. The following will describe the underlying concepts of each meta-class in detail. A `ResourceDemandingSEFF` is a special type of `ServiceEffectSpecification` [RFB04] and additionally inherits from `ResourceDemandingBehaviour`. Therefore, first these two classes will be explained.

Service Effect Specification Models the effect of invoking a specific service of a basic component [RFB04]. Therefore, it references a `Signature` from an `Interface`, for which the component takes a `ProvidedRole`, to identify the described service. This class is abstract and SEFFs for specific analysis purposes need to inherit from this class. A `BasicComponent` may have an arbitrary number of SEFFs. It can have multiple SEFFs of a different type for a single provided service. For example, one SEFF can express all external service calls with no particular order, while another one includes a restricted order, or still another one expresses resource demands of the service.

While different SEFF types have been proposed, the only type currently included in the meta-model is the `ResourceDemandingSEFF` for performance prediction. Different types of SEFFs should not contradict each other if the languages are equally powerful. For example, the order of allowed external service calls should be the same for each SEFF type modelling sequences of such calls if the modelling languages have the same expressiveness.

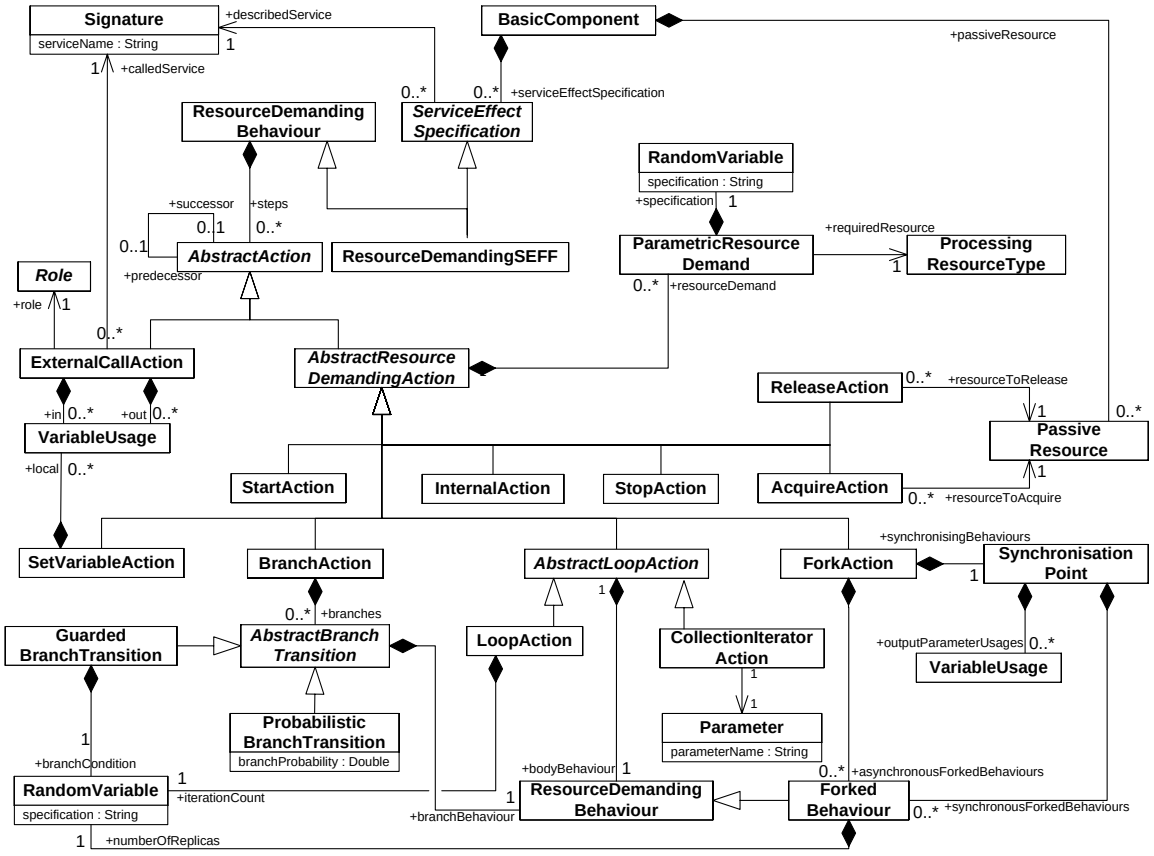


Figure 4.8: Resource Demanding SEFF (meta-model)

SEFFs are part of a component and not part of an interface, because they are implementation dependent. The SEFFs of a `CompositeComponent` are not represented in the meta-model and can be derived automatically by connecting the SEFFs of the encapsulated components of its nested `AssemblyContexts`. Different SEFFs of a single component access the same component parameter specifications. That means that parameter dependencies to the same component parameters in different SEFF types refer also to the same characterisations.

Resource Demanding Behaviour Models the behaviour of a component service as a sequence of internal actions with resource demands, control flow constructs, and external calls. Therefore, the class contains a chain of `AbstractActions`. The emphasis in this type of behaviour is on the resource demands attached to internal actions, which mainly influence performance analysis (details follow later).

Each action in a `ResourceDemandingBehaviour` references a predecessor and a successor action. Exceptions are the first and last action, which do not reference a predecessor and a successor respectively. A behaviour is valid, if there is a continuous path from the first to last action, which includes all actions. The chain must not include cycles.

To specify control flow branches, loops, or forks, component developers need to use special types of actions, which contain nested inner `ResourceDemanding-Behaviours` to specify the behaviour inside branches or loop bodies. Any `ResourceDemandingBehaviour` can have at most one starting and one finishing action.

`AbstractActions` model either a service's internal computations or calls to external (i.e., required) services, or describe some form of control flow alteration (i.e., branching, loop, or fork). The following first clarifies the notions of internal and external actions, whose meta-classes both inherit from `AbstractAction`:

Internal Action Combines the execution of a number of internal computations by a component service in a single model entity. It models calculations inside a component service, which do not include calls to required services. For a desired high abstraction level, an RDSEFF has only one `InternalAction` for all instructions between two calls to required services. A high abstraction level is needed to keep the model tractable for mathematical analysis methods. However, in principle it is also possible to use multiple `InternalActions` in direct succession to model on a lower abstraction level and enable more accurate predictions.

To express the performance-relevant resource interaction of the modelled computations, an `InternalAction` contains a set of `ParametricResourceDemands` (described below), each directed at a specific `ProcessingResourceType`. An `InternalAction` may have at most one resource demand for each `ProcessingResourceType`. Analysis or simulation tools sequentialise the set of demands and put load on the referenced resources one after another.

`InternalActions` provide an abstraction from the complete behaviour (i.e., control and data flow) of a component service, as they can hide different possible control and data flows not affecting external service calls and express their resource demands as a single stochastic expression (cf. Chapter 3.3.6). This abstraction underlies the assumption that the resource demands of a number of instruction can be captured sufficiently accurate enough in one such expression.

Parametric Resource Demand Specifies the amount of processing requested from a certain type of resource in a parametrised way. It assigns the demand specified as a `RandomVariable` to an abstract `ProcessingResourceType` (e.g., CPU, hard disk) instead of a concrete `ProcessingResourceSpecification` (e.g., 5 Ghz CPU, 20 MByte/s hard disk). This keeps the RDSEFF independent from a specific resource environment, and makes the concrete resources replaceable to answer sizing questions.

The demand's unit is equal for all `ProcessingResourceSpecifications` referencing the same `ProcessingResourceType`. It can for example be "WorkUnits" for CPUs [Smi02] or "BytesRead" for hard disks. Each `ProcessingResourceSpecification` contains a processing rate for demands (e.g., 1000 WorkUnits/s, 20 MB/s), which analysis tools use to compute an actual timing value in seconds. They use this timing value for example as the service demand on a service center in a queueing network or the firing delay of a transition in a Petri net. As multiple component services might request processing on the same resource, these analytical or simulation models allow determining the waiting delay induced by this contention effect.

Besides this parameterisation over different resource environments, `ParametricResourceDemands` also parameterise over the usage profile. For this, the stochastic expression specifying the resource demand can contain references to the service's input parameters or the component parameters. Upon evaluating the resource demand, analysis tools use the current characterisation of the referenced input or component parameter and substitute the reference with

this characterisation in the stochastic expression. Solving the stochastic expression, which can be a function involving arithmetic operators (Chapter 3.3.6), then yields a constant or probability function for the resource demand.

As an example for solving the parameterisation over resource environment and usage profile, consider an RDSEFF for a service implementing the bubblesort algorithm. It might include a CPU demand specification of $n^2 + 2000$ WorkUnits derived from complexity theory (n^2) and empirical measurements (2000). In this case n refers to the length of the list the algorithm shall sort, which is an input parameter of the service. If the current characterisation of the list's length is 100 (as the modelled usage profile), analysis tools derive $100^2 + 2000 = 12000$ WorkUnits from the specification, thus resolving the usage profile dependency. If the CPU `ProcessingResourceSpecification` the service's component is allocated on then contains a processing rate of 10000 WorkUnits/s, analysis tools derive an execution time of 12000 WorkUnits $/10000$ WorkUnits/s = 1.2 s from the specification, thus resolving the resource environment dependency.

The stochastic expression for a `ParametricResourceDemand` depends on the implementation of the service. Component developers can specify it using complexity theory, estimations, or measurements. However, how to get data to define such expressions accurately is beyond of the scope of this thesis. Woodside et al. [WVCB01] and Krogmann [Kro07] present approaches for measuring resource demands in dependency to input parameters. Meyerhoefer et al. [ML05] and Kuperberg et al. [KB07] propose methods to establish resource demands independent from concrete resources. For the scope of this thesis, it is assumed that these methods have been applied and an accurate specification of the `ParametricResourceDemand` is available.

ExternalCallAction Models the invocation of a service specified in a required interface. Therefore, it references a `Role`, from which the providing component can be derived, and a `Signature` to specify the called service. `ExternalCallActions` model synchronous calls to required services, i.e., the caller waits until the called service finishes execution before continuing execution itself. The PCM allows modelling asynchronous calls to required services by using an `ExternalCallAction` inside a `ForkedBehaviour` (described later).

`ExternalCallActions` do not have resource demands by themselves. Component developers need to specify the resource demand of the called service in the RDSEFF of that service. The resource demand can also be calculated by analysing

the providing component. This keeps the RDSEFF specification of different component developers independent from each other and makes them replaceable in an architectural model.

`ExternalCallActions` may contain two sets of `VariableUsages` specifying *input* parameter characterisations and *output* parameter characterisations respectively. `VariableUsages` for input parameters may only reference IN or INOUT parameters of the call's referenced signature. The random variable characterisation inside such a `VariableUsage` may be constants, probability distribution functions, or include a stochastic expression involving for example arithmetic operations. The latter models a dependency between the current service's own input parameters and the input parameters of the required service.

All characterisations for input parameters of a required service form a part of the usage profile of that service. The dependencies between input parameter characterisation of the calling service and input parameter characterisation of the called service are not unique for a specific usage context, but are valid for all possible usage contexts. This kind of specification is more expressive as for example the specification approach by Hamlet et al. [HMW04], where such a dependency always needs to be measured for specific test cases.

`VariableUsages` for output parameters assign variable characterisations returned by the external call to local variable names. When different `ExternalCallActions` to the same required service occur, each of them can use unique local variable names to memorise the different output parameter characterisations. This enables referring to the output parameters of different external service calls to the same service unambiguously in the RDSEFF afterwards.

The characterisations referenced in the stochastic expressions of the right hand side of such assignments may only be from OUT parameters, INOUT parameters, or return values (keyword: `serviceName.RETURN`) of the called service's signature, but not from IN parameters. The RDSEFF of the external service will set the specifications of those characterisations. Notice, that it is not the RDSEFF of the calling service, which sets those characterisations. However, it is possible to specify stochastic expressions involving arithmetic operations on such characterisations. These operations then model calculations inside the calling service performed after executing the external call, but not calculations performed in the called service.

If the system deployer allocates the respective components of the calling service and the called service to different `ResourceContainers`, an `ExternalCallAction` automatically produces load on the network device connecting these

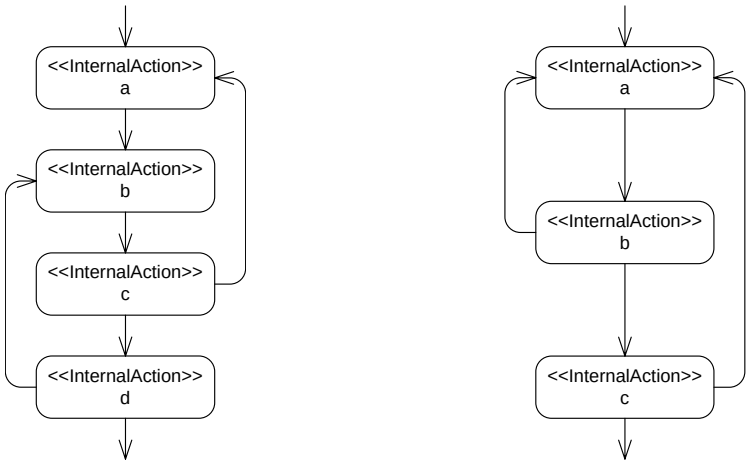
`ResourceContainers`. Analysis tools can calculate the amount of data transferred over the network by summing up all input or output parameter characterisations, which refer to the byte size of a parameter. The RDSEFF needs no additional specification of the transferred data's byte size.

Set Variable Action Assigns a variable characterisation to an OUT parameter, INOUT parameter, or return value of the service. It ensures that performance-relevant output parameter characterisations of a component service are specified to use them to parameterise the calling RDSEFF. A `SetVariableAction` must only use output parameters on the left hand side of the assignment and must not use input parameter or local variable names, because input parameters cannot be returned and local names should not be exposed to adhere the black box principle. The action is only intended to allow proper data flow modelling (i.e., output parameter passing) between different component services, but not to reveal additional internals of the service the current RDSEFF models. Thus, the assigned characterisation is not accessible in subsequent actions of the current RDSEFF.

Notice, that the stochastic expression used in this assignment must characterise the result of the whole computation of the current service. For non-trivial components, this requires a substantial stochastic approximation based on manual abstraction. However, recall that not the actual result of a component service needs to be specified, but only its performance-relevant attributes. For example, to model the return value of a component service compressing a file, using its file size divided by the compression factor as the stochastic expression is usually sufficient, while the value of the compressed file is not of interest in a performance model.

Multiple `SetVariableActions` assigning to the same output parameter might occur at different locations of the control flow in an RDSEFF. In the case of sequences, loops, and fork, the *last* assignment overwrites the former assignments and gets transferred back to the calling RDSEFF. Therefore, analysis tools may ignore the former assignments. In the case of using a `SetVariableAction` in two different branches of a `BranchAction`, only the assignment in the chosen branch is valid and gets transferred back to the caller.

Control Flow The RDSEFF defines the control flow between internal and external actions with the predecessor/successor relationship between `AbstractActions` to model sequential executions. Additionally, special actions for branching, loops, and forks allow other kinds of control flow. Other than flowcharts or UML activity



(a) Intertwined Control Flow

(b) Loop with Multiple Entry Points

Figure 4.9: The Problem of Backward References

diagrams, the RDSEFF language (as well as the usage model language) requires developers to make the branching, loop, fork bodies explicit using nested `ResourceDemandingBehaviours`.

It disallows backward references in the chain of `AbstractActions`, which are basically goto statements and can lead to ambiguities and difficult maintainability. For example, this might lead to intertwined control flows as in the example in Fig. 4.9(a), where both the sequences 'abcabcdbcd' and 'abcdbcabcd' could be occur if each backward reference is executed once, which might lead to different execution times.

Backward references also allow the specification of loops with multiple entry points as in Fig. 4.9(b). This is not desirable, as the number of loop iterations cannot be specified directly in these cases, which is however necessary for accurate performance prediction. If a developer would specify that each backward link in Fig. 4.9(b) is executed only once, both sequences 'ababc' and 'abcababc' would be possible although they would have different execution times, as 'a' is executed three times in the latter case.

To avoid such ambiguities, control flow in the PCM RDSEFF and usage model must be specified without backward references in the chain of `AbstractActions`. Branches, loops, forks, and their respective bodies have to be made explicit in the specification using nested `ResourceDemandingBehaviours` (also see Fig. 4.10 for an example in concrete syntax).

Branch Action Splits the RDSEFF control flow with an XOR-semantic, meaning that the control flow continues on exactly one of its attached `AbstractBranchTransitions`. The RDSEFF supports two different kinds of branch transitions, `GuardedBranchTransitions`, and `ProbabilisticBranchTransitions`. RDSEFFs do not allow to use both kinds of transitions on a single `BranchAction`. Analysis or simulation tools must select exactly one transition based on the included guard or probability, before continuing at a `BranchAction`.

Guarded Branch Transition Provides a link between a `BranchAction` and a nested `ResourceDemandingBehaviour`, which includes the actions executed inside the branch. It uses a guard, i.e. a boolean expression specified by a `RandomVariable`, to determine whether the transition is chosen. If the guard evaluates to true, the branch is chosen, otherwise if the guard evaluates to false another branch transition must be chosen.

The guard may contain references to the service's input parameters or component parameters. A component developer can specify complex boolean expressions by using the `AND`, `OR`, and `NOT` operations provided by the StoEx framework. As the domain expert may have characterised the parameters used in a guard with probability distributions, it might happen that a guard does not evaluate to true or false with a probability of 1.0. For example, the specification can express that a guard evaluates to true with a probability of 0.3, and to false with a probability of 0.7. In any case, the probabilities of the individual guards attached to all `GuardedBranchTransitions` contained in a `BranchAction` must sum up to 1.0.

There is no predefined order in evaluating the guards attached to a `BranchAction`. This differs from programming languages such as C or Java, where the conditions on if/then/else statements are evaluated in the order of their appearance in the code. Such programming languages allow overlapping branching conditions (for example, `if (X<10) //... else if (X<20) // ...`), which are not allowed for the guards in `GuardedBranchTransitions`, because the missing order specification would lead to ambiguous boolean expressions and enable more than one guard to become true. If `X` would have the value 5, both conditions would evaluate to true if they would be used directly as guards in `GuardedBranchTransitions`. The correct specification of the guards in this case would be `X.VALUE` < 10 and `X.VALUE` $\geqslant 10$ `AND X.VALUE` < 20.

Guards might lead to stochastic dependencies when evaluating variable charac-

terisations inside a branched behaviour. For example, if the guard `X.VALUE` < 10 had formerly evaluated to true, and the RDSEFF uses `X.VALUE` inside the branched behaviour, the sample space of the random variable specifying the characterisation must be restricted, as the event that `X` takes a values greater than 10 cannot occur anymore. Therefore its probability is zero. Any variable characterisation always needs to be evaluated under the condition that all guards in the usage scenario's path to it have evaluated to true.

Probabilistic Branch Transition Like a `GuardedBranchTransition`, this transition provides a link between a `BranchAction` and a nested `ResourceDemandingBehaviour`, which includes the actions executed inside the branch. But instead of using a guard, it specifies a branching probability without parameter dependencies. Analysis tools may directly use it to determine the transition where the control flow continues. The probabilities of all `ProbabilisticBranchTransitions` belonging to a single `BranchAction` must sum up to 1.0.

Although a probabilistic choice at a branch usually does not happen in a computer program, `ProbabilisticBranchTransitions` provide a convenient way of modelling in case the actual parameter dependency is too hard to determine or too complex to integrate into a guard. It can also be useful for newly designed components, where the parameter dependency on the control flow guard is still be unknown.

However, this construct potentially introduces inaccuracies into the performance model, because it does not reflect the influence of input parameters. Therefore, predictions based on this model can be misleading, if the used input parameters would result in different branching probabilities. The component developer cannot foresee this, when specifying the RDSEFF using `ProbabilisticBranchTransitions`.

Fork Action Splits the RDSEFF control flow with an AND-semantic, meaning that it invokes several `ForkedBehaviours` concurrently. This action is subject to research by Happe [Hap08], therefore the following description might be outdated. `ForkActions` allow both asynchronously and synchronously forked behaviours.

Synchronously `ForkedBehaviours` execute concurrently and the control flow waits for each of these behaviours to terminate before continuing. Each `ForkedBehaviour` can be considered as a program thread. All parameter characterisations from the surrounding RDSEFF are also valid inside the `ForkedBehaviours` and can be used to parameterise resource demands or control flow constructs. The pa-

rameter characterisations are the same in each `ForkedBehaviour`. Component developers can use a `SynchronisationPoint` to join synchronously `ForkedBehaviours` and specify a result of the computations with its attached `VariableUsages`.

Asynchronously `ForkedBehaviours` also execute concurrently, but the control flow does not wait for them to terminate and continues immediately after their invocation with the successor action of the `ForkAction`. Therefore, there is no need for a `SynchronisationPoint` in this case. It is furthermore not possible to refer to results or output parameters of asynchronously `ForkedBehaviours` in the rest of the RDSEFF, as it is unclear when these results will be available.

Loop Action Models the repeated execution of its inner `ResourceDemandingBehaviour` for the loop body. The number of repetitions is specified by a random variable evaluating to integer or an `IntPMF`. The number of iterations specified by the random variable always needs to be bounded, i.e., the probabilities in an `IntPMF` for iteration numbers above a certain threshold must be zero [KF06]. Otherwise, it would be possible that certain requests do not terminate, which would complicate performance analyses.

The stochastic expression defining the iteration random variable may include references to input or component parameters to model dependencies between the usage profile and the number of loop iterations. Notice, that loop actions should only be modelled if the loop body contains either external service calls or resource demands directed at special resources. Otherwise, control flow loops in component behaviour should be abstracted by subsuming them in `InternalAction`, which combine a number of instructions. The influence of different iterations length of such internal loops need to be reflected stochastically by the random variable specifying the `ParametricResourceDemand` of that `InternalAction`.

Other than Markov chains, RDSEFFs do not specify control flow loops with an re-entrance and exit probability on each iteration. Such a specification binds the number of loop iterations to a geometrical distribution, which reflects reality only in very seldom cases. For example, a loop with a re-entrance probability of 0.9 and an exit probability of 0.1, would result in 1 loop iteration with a probability of 0.9, 2 loop iterations with a probability of $0.9 * 0.9 = 0.81$, 3 loop iterations with a probability of $0.9 * 0.9 * 0.9 = 0.729$, and so on. But in many practical cases, the number of iterations is a constant, or the probability for higher iteration numbers is higher than for lower ones. This cannot be expressed directly via a Markov chain (also see [DG00]).

Inside the ResourceDemandingBehaviour of LoopActions, it is assumed that random variables are stochastically independent. This is not true in reality, and for example leads to wrong predictions if the same random variable is used twice in succession inside a loop body. In this case, the second occurrence is stochastically dependent to the first occurrence, as the value does not change between two occurrences. Therefore, component developers should be aware of such inaccuracies when using random variables twice inside the body behaviour of a LoopAction.

Collection Iterator Action Models the repeated execution of its inner ResourceDemandingBehaviour for each element of a collection data type. Therefore it contains a reference to an input parameter of the service's signature, which must be of type CollectionDataType. The NUMBER_OF_ELEMENTS must be specified from the outside of the component, either by another RDSEFF or by an usage model calling this service. It can be of type integer or IntPMF.

Besides the source of the number of iterations, CollectionIteratorActions differ from LoopAction only in their allowed stochastic dependence of random variables inside the loop body's ResourceDemandingBehaviour. If the same random variable occurs twice in such a loop body, analysis tools must evaluate the second occurrence in stochastic dependence to the first occurrence. This complicates the involved calculation and might lead to the intractability of the model, therefore component developers should use CollectionIteratorActions with care and only include them if they expect that the prediction results would be vastly inaccurate without it.

Passive Resource BasicComponents can contain a number of PassiveResources, such as semaphores, thread pools, database connection pools, etc. (also see [Hap08]). Passive resources can for example encapsulate the access to critical areas. They contain a limited number of items, which have to be acquired to carry out certain calculations, and later be released again.

In an RDSEFF, component developers can specify an AcquireAction, which references a passive resource types. Once analysis tools execute this action, they decrease the amount of items available from the referenced passive resource type by one, if at least one item is available. If none item is available, because other, concurrently executed requests have acquired all of them, analysis tools enqueue the current request (first-come first-serve scheduling policy) and block it's further execution. One of the other concurrent requests then needs to execute a Release-

`Action`, which then again increases the number of available item for the given passive resource type, before the current request can continue.

Acquisition and release of passive resources happen instantaneously and do not consume any time except for waiting delays before actual acquisition. Resource locking may introduce deadlocks when simulating the model, however, for performance analysis with the PCM it is assumed that no deadlocks occur. Otherwise, the model first needs to be fixed accordingly before carrying out the performance prediction.

4.3.2 Example

The example in Fig. 4.10 illustrates a simple RDSEFF instance. It uses a concrete syntax similar to UML activity diagrams. Each graphical element denotes its meta-class with an included stereotype, i.e., the name of the meta-class enclosed in angle brackets. Although the graphical visualisation resembles annotated UML activity diagrams for higher intuition, there is no relation to the UML meta-model. Using the abstract syntax, i.e., an object graph of the meta-classes, would result in a more complex and less intuitive visualisation of the model.

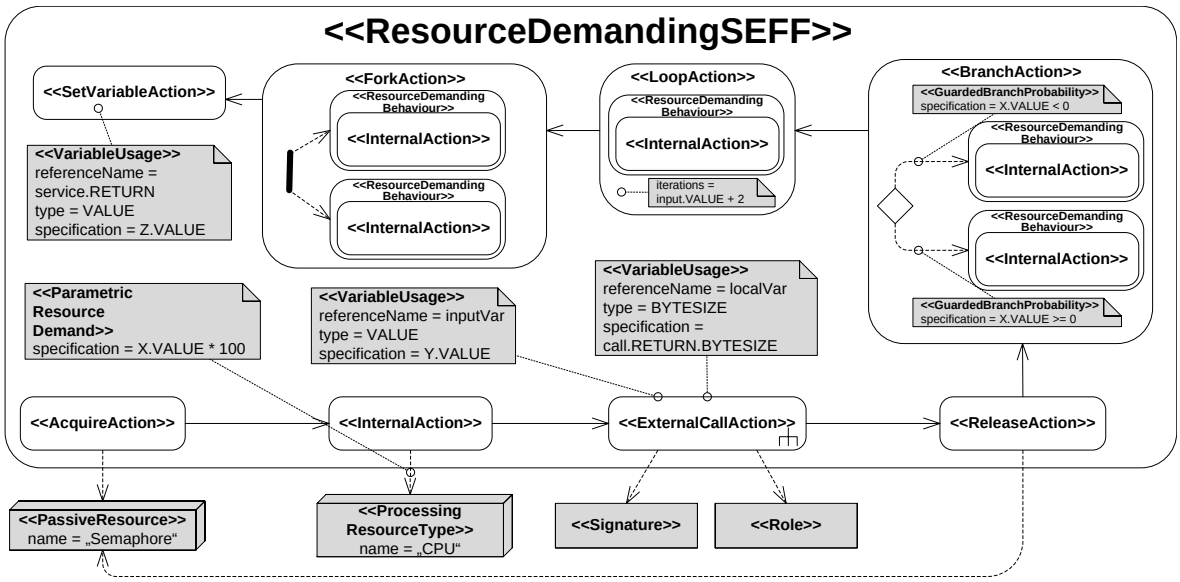


Figure 4.10: RDSEFF (Example)

The figure shows most of the formerly introduced RDSEFF elements condensed in a single model instance. The RDSEFF's control flow starts in the lower left corner with an `AcquireAction`. It references a `PassiveResource` named "Semaphore",

i.e., the component service requests an instance of this resource when executing this action.

Once the service has obtained the semaphore, it executes an `InternalAction`. It references a `ProcessingResourceType` named CPU. The service requests processing by the CPU. The RDSEFF only refers to the resource type, but not to a concrete resource instance. When the system deployer later allocates the component of the service to a `ResourceContainer`, the `ProcessingResourceSpecification` inside that `ResourceContainer`, which references the same `ProcessingResourceType` is used by the `InternalAction`.

All resource types and passive resources must already be available (i.e., specified and referable) to the component developer, when defining an RDSEFF. For `ProcessingResourceTypes`, this requires the availability of a `ResourceTypeRepository`, whose contents are agreed on between component developers and system deployers. Component developers specify `PassiveResources` as part of a `BasicComponent`. Therefore, a `BasicComponent` with the provided `PassiveResources` needs to be available before defining an RDSEFF referencing it.

The `InternalAction` in the example uses a `ParametricResourceDemand` to specify the requested amount of processing from the referenced resource type. This resource demand is parametric both for the usage profile (it can reference input parameter characterisations) and the resource environment (it specifies platform-independent demands).

An `ExternalCallAction` follows the `InternalAction`. It references the called service's `Signature`, which is part of an `Interface` (Chapter 3.2.2), and a `RequiredRole`, which is part of the component specification (Chapter 3.2.2). During later composition, the software architect will bind the referenced `RequiredRole` to the `ProvidedRole` of another component using an `AssemblyConnector`. Then, the RDSEFF invoked by this `ExternalCallAction` can be determined by following the `AssemblyConnector` to determine the connected component and retrieving its RDSEFF for the referenced `Signature`.

In the example, the component developer has attached two `VariableUsages` to the `ExternalCallAction`. The first specifies an input parameter characterisation. It expresses, that "VALUE" of the parameter "inputVar" from the referenced `Signature` is characterised as "Y.VALUE". "Y" can be of any data type and its "VALUE" can for example be characterised with a probability distribution (Chapter 3.3). The second `VariableUsage` is an output parameter characterisation. The introduced variable "localVar" get assigned with a variable characterisation pro-

duced by the called RDSEFF. ”localVar.BYTESIZE” is set to the byte size of the called service’s return value (”call.RETURN.BYTESIZE”). Thus, the component developer can later use ”localVar” in the RDSEFF to reference to this characterisation.

After the `InternalAction` and `ExternalCallAction`, a `ReleaseAction` frees the formerly acquired `PassiveResource`. Therefore, the model expresses that the service no longer holds the corresponding semaphore, which is now again available for acquisition by other services.

The `ReleaseAction` is followed by a chain of different control alterations. First, the `BranchAction` splits the control flow with an XOR semantic, i.e., it executes exactly one of the included behaviours. The choice depends on the `GuardedBranchTransition`, which include a parameter dependency and check whether the value of input parameter ”X” is less or greater than zero. The resource demands of the `InternalActions` within the branched behaviours have been omitted in this figure for clarity.

Following the `BranchAction`, the `LoopAction` models a repetitive execution of the included behaviour. The component developer has specified the number of iterations with a dependency to an input parameter (”input.VALUE+2”). Alternatively, component developers may use `CollectionIteratorActions`, which also iterate over their inner behaviour, but reference an input parameter of a collection data type, and execute one iteration for each element of the collection.

The following `ForkAction` splits the control flow with an AND semantic, i.e., it executes each of the included behaviours concurrently. In this case the invocation of the inner behaviours is asynchronous and the service directly continues with the following `SetVariableAction` and does not wait for the inner behaviours to terminate. The `SetVariableAction` finally characterises the return value of the modelled service and assigns ”Z.VALUE” as the specification of its random variable.

4.3.3 Comparison to Related Work

As already discussed in Chapter 2.5.1, there are several other component-based performance prediction approaches, which all feature their own component performance specification languages. As the RDSEFF language has now been described in detail, it is possible to conduct a more detailed comparison to these approaches.

As a main contribution of this work are the parameter dependencies introduced into the RDSEFF language, Table 4.1 contains the capabilities of the different modelling languages to express such dependencies. CB-SPE and Hamlet do not include

a notion of service parameters declared in interfaces and therefore do not model parameter dependencies. However, these approaches allow changing attributes of the resulting performance model and consider this as parameterisation. CB-SPE is based on the UML-SPT profile, which does not model service parameters.

Name	CB-APPEAR	CBML	CB-SPE	Hamlet	RESOLVE-P	ROBOCOP	PALLADIO
Literature	[EF04, EFH04]	[Wu03, WMW03, WW04]	[BM04a, BM04b]	[HMW01, HMW04]	[SKK01]	[BdWCM05, BCdW06a, BCdW06b, BCdK07]	[BKR08]
Resource Demand	✓	(✓)	-	-	(✓)	✓ (constant)	✓
Branch Probability	(✓)	-	-	-	-	-	✓
Number of Loop Iterations	(✓)	-	-	-	-	(✓)	✓
Number of Forked Branches	-	-	-	-	-	-	-
Input Parameter	-	-	-	-	-	✓ (constant)	✓
Output Parameter	-	-	-	-	-	-	✓
External Call Input	-	-	-	-	-	✓ (constant)	✓
External Call Output	-	-	-	-	-	-	✓
Internal State	-	-	-	-	-	- (Comp. Par.)	- (Comp. Par.)
Passive Resource Capacity	-	✓	-	-	-	-	✓
Number of Component Replications	-	✓	-	-	-	-	-

Table 4.1: Service Parameter Dependencies in CB-Performance Prediction Methods

CB-APPEAR does include resource demands parameterised over service inputs. The authors also use parameterised branch conditions and loop iteration number, however only in an ad-hoc manner without a strict deviation between component developer and software architect. CBML allows changing attributes of the resulting performance model and explicitly supports component parameters for the capacity of passive resources and the number of replications. However, as the slots representing interfaces in this approach do not contain a notion of service parameters, no dependencies to such parameters can be specified.

The ROBOCOP performance prediction approach supports explicit parameter dependencies on resource demands, loop iteration numbers, and input parameters. It does not support output parameter dependencies, and branch conditions referring to parameter values. Furthermore, it only uses constant integers to specify parameter values, whereas the PCM supports random variables with arbitrary distribution functions for more expressiveness.

4.3.4 Limitations

There are still several limitations that restrict the modelling capabilities of the RD-SEFF language and therefore the prediction accuracy of the resulting models:

- **Limited Internal State:** As formerly discussed, internal component state can heavily influence performance properties in certain situations. The PCM uses

an abstraction from actual internal state in form of the so-called component parameters. These parameters are constant across different usage scenarios and equal for all users to avoid state space explosion. This abstraction is useful to model several practically relevant situations as demonstrated in the case study in Chapter 7.3. However, a more accurate modelling of internal state is necessary for other scenarios.

- **No Scopes for Stochastic Dependencies:** The RDSEFF language and the corresponding analysis algorithms assume stochastic independence between the random variables used in a RDSEFF. However, if an RDSEFF uses a random variable for a parameter characterisation twice (e.g., for two external calls), both uses are stochastic dependent on each other, because the parameter value does not change between two occurrences. Therefore, performance prediction in such situations leads to inaccurate results with the current PCM version. The `CollectionIteratorAction` has been introduced to circumvent this problem for loop bodies. Its concept could be generalised to allow arbitrary scopes in RDSEFFs, where random variables are evaluated assuming stochastic dependence. However, this might lead to state space explosion and requires further research.

- **No Component-Internal Reuse:** Multiple services of a single software component might internally reuse functionality of the component. This is comparable to private methods in object-oriented programming, which can be accessed by different methods inside a class. RDSEFFs do not support such component-internal subroutines, because they try to abstract from component internals as much as possible. To model the performance influence of such internal functionality, the respective `InternalActions` have to be replicated in the RDSEFFs of each of the component services using them. It might be useful to introduce concepts to reuse such internal functionality, so that it has to be modelled only once. However, this might result in a too low abstraction level in the model instances, so that they expose too much of the component internals, which violates the black-box principle.

- **Limited Component-internal Concurrency:** As already mentioned, modelling component-internal concurrency is still immature in the RDSEFF modelling language. So far, it supports several low-level construct, such as synchronous and asynchronous forked behaviours. However, it is unclear how

data dependencies between these forked behaviours can be expressed sufficiently. For the future, it would also be desirable to have higher level modelling constructs resembling well-known concurrency patterns. This could substantially lower the effort to model complex concurrency situations.

- **No Support for Pro-Activity:** The PCM follows a synchronous call-and-return semantic for calls between components. Every component has to be invoked from the outside and it is not possible that component service call other components autonomously. However, such pro-active components do exist in reality, therefore RDSEFFs should include constructs to model them.

- **No Support for Memory Consumption:** RDSEFFs only support resource demands to processing resources, such as CPUs or hard disk, and to passive resources, such as semaphores. There is no support for specifying memory consumptions by a component service. It is desirable to make predictions for the memory consumption of a component-based software architecture, so that the hardware can be sized accordingly.

- **Other QoS attributes:** While RDSEFFs so far focus on performance modelling, other compositional QoS attributes, such as reliability and availability could be included into the modelling language. It would further be possible to conduct combined prediction for performance and reliability to assess the performability of a component-based system.

Overcoming the limitations of RDSEFFs listed above involves extending the language. This should be done with care as a higher complexity of the language might also lead to more complex model instances, which could become difficult to simulate or intractable for mathematical analyses. Furthermore, the component principles of independent deployment and information hiding should be adhered to when adding new features. The tradeoff between the complexity of the language and the resulting models versus the accuracy of predictions needs to be analysed with care.

Notice, that the PCM has further limitations (e.g., static architectures, limited network support, etc.), which are not listed here, because they do not concern the RDSEFF language but other parts of the model, such as the `System` or `Resource-Environment`.

4.3.5 Discussion

After the description of the RDSEFF's core concepts, an example, and a list of limitations, the following discusses some RDSEFF features and properties:

Grey-Box View RDSEFFs induce a grey-box specification of a software component (also see Chapter 2.1.2). On one hand, they provide additional information about component internals besides the interfaces and are thus no black-box specification. On the other hand, they abstract from the component's source code and algorithms and are thus no white-box specification. This violation of the black-box principle, which is usually demanded from software components [SGM02], is necessary to create a reusable performance model.

However, RDSEFFs limit the additionally revealed information about component internals to elements necessary for performance analysis. They do not reveal local variables of component services, because parameter dependencies in RDSEFFs may only refer to input parameters specified in the provided interface. The algorithms implemented by the component are highly abstracted and mostly subsumed in single internal actions. An RDSEFF only reveals control flow that alters the sequence of executing required services. Therefore, the intellectual properties to the source code by a component developer are not affected when an RDSEFF is published.

Furthermore, in many cases software architects do not need to understand RDSEFFs of existing components, because they operate on a higher abstraction level of the model and only refer to components. Performance analysis tools exploit RDSEFFs and might point to components, which are a bottleneck, prompting software architects to ask the corresponding component developers to fix the components and provide new RDSEFFs.

Variable Abstraction Level The standard abstraction from the source code induced by an RDSEFF has already been described. All instructions between two external service calls should be combined into a single internal action, whose parametric resource demand should compactly express the demand of these instructions with stochastic means. However, this abstraction level is only a guidance for component developers to avoid state space explosion during performance analysis. It is not enforced by the PCM.

In contrast, component developers could use multiple internal actions in succession and model the control flow between them without having an `External-`

`Action`. As an extreme case, an internal action could be used for each code instruction, which would result in a model without an abstraction therefore being as complex as the source code. This more fine-granular modelling potentially makes the performance model more accurate. However, it can easily lead to intractable specifications and revelation of intellectual properties.

Up to this point, a better abstraction level (with higher accuracy but still tractability) than the standard abstraction level described above is unknown. More research into this direction is needed. It might depend on the component's size, which abstraction level is useful. More complex components could require a lower abstraction level to make accurate predictions. Different components could require different RDSEFFs with different abstraction levels.

Smith et al. [Smi02] have stated that the Pareto principle (aka. 80/20 rule) applies to the execution times of software. From experience, it is known that roughly 80 percent of the execution time are consumed by only 20 percent of the underlying code. Therefore, a component developer needs to identify these 20 percent and provide less abstract RDSEFF parts for them, while abstracting the other 80 percent as much as possible. Empirical studies need to investigate whether this is feasible. Profiling techniques used for RDSEFF generation could support the step.

Information Sources A lot of information is necessary to create an RDSEFF accurately modelling the performance properties of a component service. It is debatable whether such a complex model is useful in practice, where developers often cannot spend much time on performance modelling. However, complete manual modelling of RDSEFFs should only be necessary if a component developer designs a new component. For already implemented components, RDSEFFs should be generated as far as possible from existing development artifacts, such as functional design documents, code, test cases, and performance measurements. Chapter 5 sketches a hybrid approach combining static code analysis and profiling techniques to generate RDSEFFs from code. A static code analysis to partially create RDSEFFs has been implemented as part of this thesis (also described in Chapter 5).

Third-Party Reuse RDSEFFs are specifically designed for third-party reuse. Component developers shall submit RDSEFFs of their components to public repositories, where software architects retrieve them to build system models. While this process model is theoretically advantageous over classical approaches without reuse, its success in practise still needs to be proved. Marketplaces for software compo-

nents have long been envisioned and researched [SGM02], however until today the software industry has not realised them. It is therefore questionable whether public repositories for component performance models can be achieved in practise in the near future.

4.4 Mapping PCM instances to QPNs

The following introduces a mapping from the PCM to Hierarchical Queueing Petri Nets (HQPN) [BBK94] to formally specify the performance-related behavioural semantics of PCM instances. Chapter 4.4.1 first motivates the use of HQPNs, before explaining the general structure of the HQPN resulting from a transformation of a PCM instance and listing basic assumptions of the mapping. Chapter 4.4.2 then describes the mapping of PCM usage model fragments to HQPN fragments and provides a comprehensive example. Finally, Chapter 4.4.3 describes the mapping of PCM RDSEFF fragments to HQPN fragments and also illustrates the mapping to an example.

4.4.1 Introduction

Motivation The model transformation described in the next subsections uses HQPNs [BBK94] as target model. It is intended to give performance-related concepts in the newly introduced behavioural modelling languages formal semantics. The mapping does not give formal semantics to component-related concepts, such as PCM interfaces or PCM roles, because they have no counterparts in the performance domain.

Appendix B.2 provides a step-wise introduction into HQPNs and includes formal definitions of different kinds of Petri nets, on which the HQPN definition (Def. 21) is based on. The following assumes that the reader is familiar with the HQPN formalism.

HQPNs directly support many concept from PCM instances, e.g., control flow, synchronisation, timing, queueing, and parameter dependencies with the Petri net colours. Therefore, it is less complicated to specify the mapping from PCM instances to HQPN instances as it would be with other formalisms like SPAs or QNs.

Although existing solvers for HQPNs (cf. [BBK95, KB06]) could be used to predict the performance of the resulting models, this has not been done in this work as an implementation of the mapping is future work. This chapter only shows that

it is possible to map PCM instances to HQPNs, which shall clarify the meaning of performance-related PCM concepts.

However, to predict the performance of PCM instances, Becker [Bec08] has implemented a model-transformation to a discrete-event simulation framework called "SimuCom". SimuCom uses similar semantics as the HQPN mapping defined in the following, but is not rooted in a specific formal performance model (e.g., SPA, SPN). Nevertheless, the Java code implementing the simulation follows the semantics presented in the following. Thus, performance predictions with SimuCom are assumed to yield the same results as using existing HQPN solvers.

General HQPN Structure for a PCM Instance Fig. 4.11 depicts the structure of the mapping from a valid PCM instance to a HQPN. The mapping transforms each `Workload`, `ScenarioBehaviour`, and `ResourceDemandingBehaviour` in a specific `AssemblyContext` into a QPN subnet. The resulting HQPN includes one layer of `Workload` subnets and an arbitrary amount of layers of `ScenarioBehaviour` and `ResourceDemandingBehaviour` subnets. Fig. 4.11 depicts only two layers for the latter.

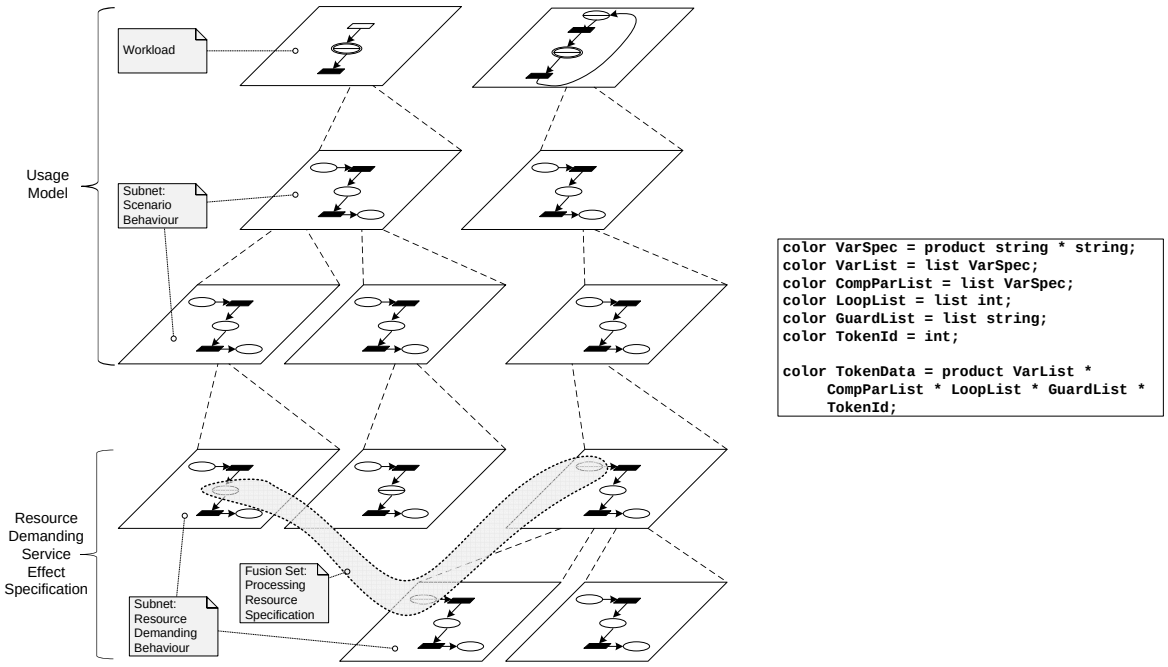


Figure 4.11: HQPN Instance as a Result of a Mapping from a PCM Instance

The HQPN represents each user in the `UsageModel` or each request inside the

`System` with a single token. The token's colour is a complex data type named `TokenData` (Fig. 4.11, right hand side), which contains:

- `VarList`: a list of currently valid parameter characterisations (as a tuple ('reference name', 'specification')) including possibly input/output characterisations
- `CompParList`: a list of currently valid parameter characterisation specified as component parameters by the domain expert, software architect, component developer.
- `LoopList`: a list of loop iteration numbers. When a token enters a loop in the usage model or in an RDSEFF, the loop iteration number is added to the list. It represents the number of iterations a token must still execute in the current loop upon each completed loop iteration.
- `GuardList`: a list of branching guards, which currently have evaluated to true. The net uses them to calculate probability distributions with stochastic dependencies.
- `TokenID`: a unique ID for each token. It is used merge tokens again after splitting and firing them into subnets.

In the following, an instance of data type `TokenData` is called `a = (varList, compParList, loopList, guardList, tokenId)`. The mapping from PCM elements to these data types preserves parameter dependencies from PCM instances in the Petri net. It includes them in the firing weights of transitions or service demands for queueing places. If a token representing a request passes a transition with a parametric firing weight, the actual weight without parameter dependencies can be determined by using the current parameter characterisations encoded in the token's colour. The functions for realising this are coded in Standard ML [MTH90], which is commonly used for coloured Petri nets [Jen92]. The following subsections will describe this in detail.

The mapping transforms `ProcessingResourceSpecifications` (i.e., CPU, HD, etc.) of a PCM instance into queueing places, which have the scheduling policies specified in the PCM instance. As different RDSEFFs access the *same* resources, the mapping uses fusion sets (Appendix B.2) to merge queueing places representing a single resource in different QPN subnets.

While the PCM supports the specification of general distribution functions, HQPNs only support exponentially timed transitions. However, the mapping uses phase-type distributions (e.g., Coxian or hyper-exponential distributions [BH07]) to approximate the general distribution functions. In the following, a function $Ph(x)$ is assumed to convert any distribution function x into a phase-type distribution, which can be used in the net. For example, to express a Coxian distribution, multiple exponential distributions (e.g., in timed transitions) need to be executed in sequence. For hyper-exponential distributions multiple exponentially timed transitions can be executed in sequence and parallel. It is assumed that this mapping does not alter the performance properties of the model significantly.

Stochastic expressions as described in Chapter 3.3.6 can contain arithmetic operations, such as addition, multiplication, etc. The mapping described in the following does not include a definition of these operations in Standard ML. Chapter 3.3.6 has already described the semantics of these operations. As the goal of the mapping is to provide semantics for PCM concepts, an implementation of the operations in the Petri net language has been omitted.

4.4.2 Usage Model Semantics

This subsection describes the mapping from usage model fragments to QPN fragments and subnets. It will subsequently explain the mapping for `ClosedWorkload`, `OpenWorkload`, `ScenarioBehaviour`, `Branch`, `Loop`, `Delay`, and `EntryLevelSystemCall`. Thus, this part of the mapping includes all classes from the meta-model (cf. Fig. 4.6) and is complete. To map a complete PCM instance to a HQPN, the individual mappings can be combined. The example presented at the end of this subsection will illustrate that.

To specify the mapping, a figure is given for each of the meta-classes with a usage model instance on the left hand side and the corresponding QPN on the right hand side. The figure uses the concrete syntax for both formalisms instead of the abstract syntax for a compact illustration.

Closed Workload The HQPN represents each workload of a PCM instance as a dedicated QPN. A `ClosedWorkload` is mapped according to Fig. 4.12. The starting place p_1 is a timed queueing place and initially contains as many tokens as specified in the PCM instance as user population ($M_0(p_1) = X$, with X being the number of users). The queue of p_1 is $q_1 = G/G/\infty/IS$, making the place a delay node with infi-

nite server scheduling. The place $p1$ simulates the user think time, which is mapped to the service demand of each user token at this place: $\mu_1(TokenData) = Ph(Y)$, where Y is the think time specification. The function $Ph(x)$ is assumed to transform any distribution function x into a phase-type distribution.

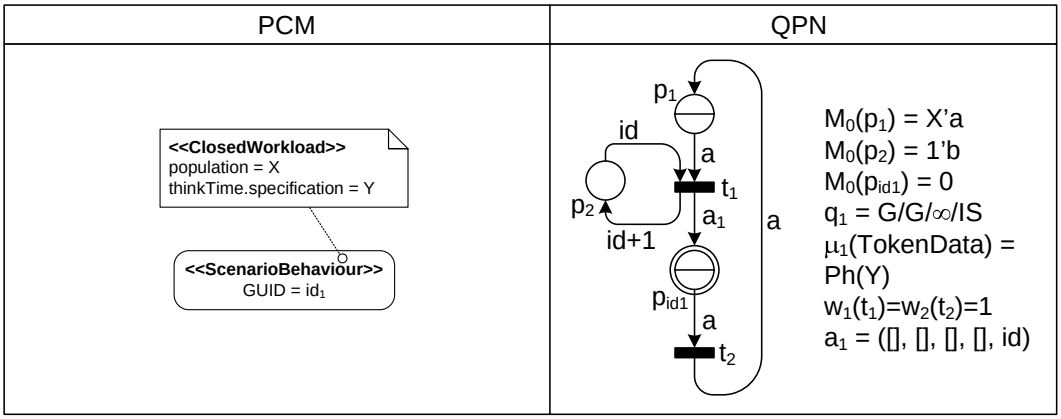


Figure 4.12: Mapping PCM2QPN: Closed Workload

The mapping adds a unique user ID id to the token in the definition of a_1. The net uses this ID later to direct tokens to different places. The id for the current token is retrieved from the place p_2, and incremented afterwards. Thus each token gets a new number. Place p_2 is part of a fusion set with all other places for user id generation in all subnets for workloads.

t_1 fires each user token into a QPN subnet place p_{id1}, which represents the `ScenarioBehaviour` directly contained in the current `UsageScenario`. Because the mapping uses GUID of the PCM's `ScenarioBehaviour` specification, it is unambiguous. After returning from the subnet, t_2 fires each token again into the timed queueing place p_1, so that it can again execute the `ScenarioBehaviour` after the specified think time.

Open Workload The mapping for an `OpenWorkload` in Fig. 4.13 is similar to the mapping for the `ClosedWorkload`, except that not a fixed number of tokens circulates in the net, but that tokens are created by t_1 and later destroyed by t_2. t_1 is a timed transition with a firing delay $Ph(X)$ derived from the inter-arrival time specification of the PCM instance. Because t_1 creates an unlimited number of tokens, this can lead to unlimited queue lengths, if one of the service times of a queueing place is longer than the inter-arrival time.

The mechanism of invoking the `ScenarioBehaviour` corresponding to the `OpenWorkload` is the same as for `ClosedWorkloads`. a_1 is the same variable in-

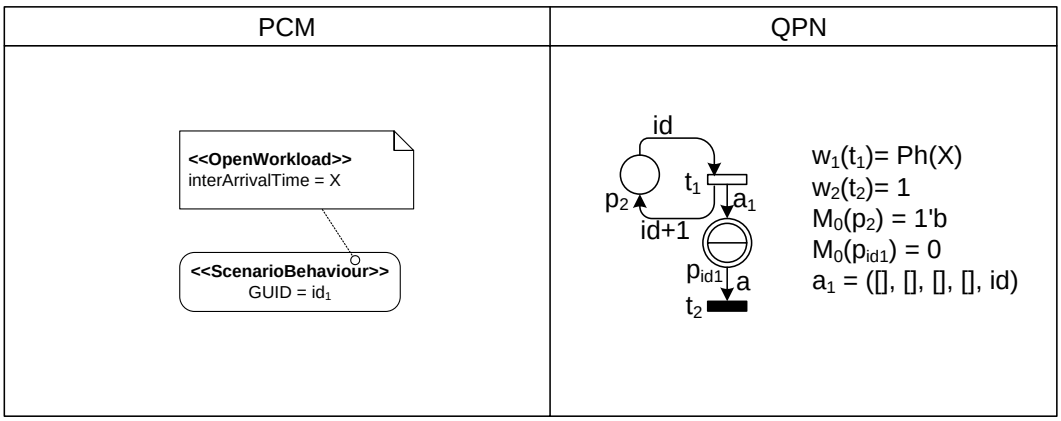


Figure 4.13: Mapping PCM2QPN: Open Workload

stance as before and initialises the variable characterisations. Opposed to `ClosedWorkloads` however, t_2 immediately destroys each token, which returns from the subnet representing the `ScenarioBehaviour`.

Notice that for all following places p (except for places representing `PassiveResources`) used in the illustrated QPNs, $M_0(p) = 0$, i.e., the initial number of tokens is zero. Only a place of the QPN for a `ClosedWorkload` contains a number of tokens initially, all other places are empty.

Scenario Behaviour Invoking a `ScenarioBehaviour` as depicted in Fig. 4.14 does not change the behaviour of the net performance-wise. It is a structuring mechanism without changing variable instances or consuming time. Each token fired into a subnet place referring to a QPN subnet via the `ScenarioBehaviour's` GUID, gets inserted into the $input$-place of the subnet. t_1 then immediately fires the token into the place p_{id2}, which represents the first action of the behaviour identified by its GUID. It can be a `Branch`, `Loop`, `Delay`, or `EntryLevelSystemCall`.

As in [BBK94], the subnet for a `ScenarioBehaviour` contains a place to count the $actual\_population$ inside the subnet. This ensures that not more tokens than entered before can be fired out of a subnet place. After a token reaches the final action of the `ScenarioBehaviour` represented by place p_{id3} in the QPN, t_2 fires the token into the $output$-place of the subnet, where it is ready to be fired into successive places of the subnet place.

Branch Mapping `Branches` with branched behaviours from a usage model instance to a QPN is straight-forward (Fig. 4.15), as there is a direct counterpart for each PCM element in the QPN. The branch probabilities p_i from the branch transi-

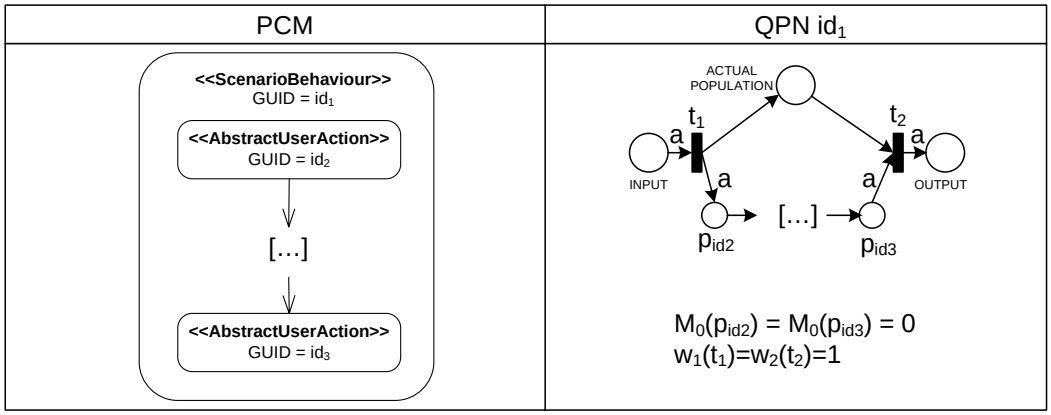


Figure 4.14: Mapping PCM2QPN: ScenarioBehaviour

tions in the PCM directly translate to the firing weights of the immediate transitions t_i, for $2 \leqslant i \leqslant n$. Each branch behaviour is represented as a subnet place with the corresponding GUID in the QPN. After completing one of the branch behaviours, t_i' fire the token into the starting place of the `Branch's` successor action ($w_i(t_i') = 1$; $2 \leqslant i \leqslant n$). None of the forward and backward incidence functions in this QPN change the variable `a`, as they pass through the current instance.

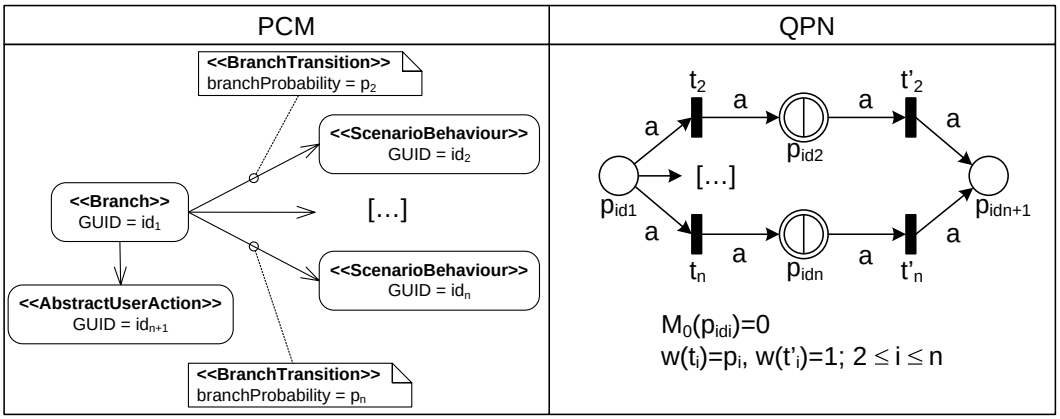


Figure 4.15: Mapping PCM2QPN: Branch

Loop Fig. 4.16 shows the mapping of a `Loop` with a body behaviour to a QPN. First, a_1 draws a sample from the loop iteration distribution function (i.e., an `IntPMF`) from the PCM instance, and adds the resulting integer to a list of loop iteration integers. This is list of integers instead of a single integer, because loop can be executed recursively nested, and the token needs to memorise all current loop

counter. The firing of the transitions t_2 and t_3 depends on whether the first element in the list of loop iteration integers (i.e., the current loop counter) is zero. If it is not zero, t_2 fires the current token into the subnet place p_id2, which indicates that the `ScenarioBehaviour` with GUID id_2 representing the loop gets executed once.

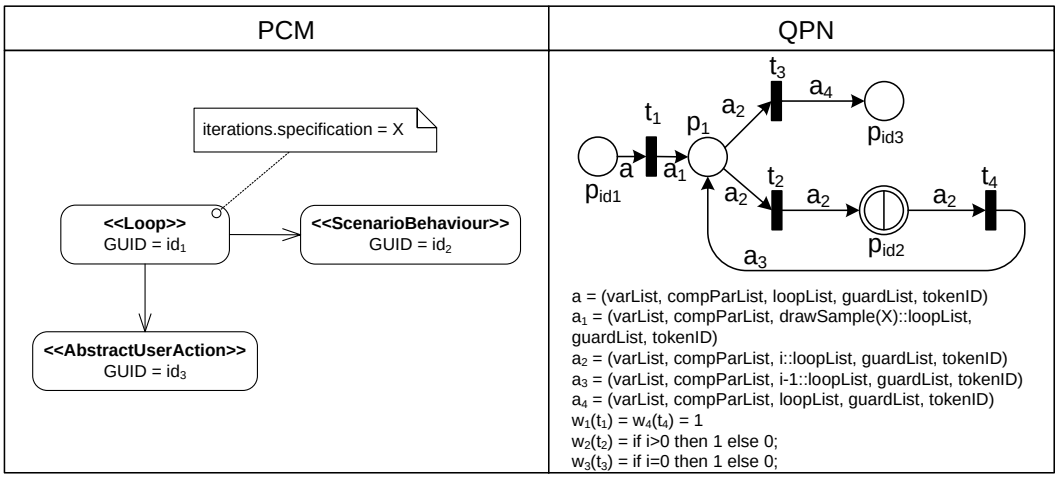


Figure 4.16: Mapping PCM2QPN: Loop

After its execution, the token returns from the subnet place and t_4 fires. Then, a_3 decrements the current loop counter by one and t_2 and t_3 again check whether the counter has reached zero. If the counter finally reaches zero, t_3 fires and a_4 removes the counter from the list of loop iteration integers. The token then is fired to the starting place representing the successor of the current `Loop` (p_id3).

Delay A timed queueing place p_{id1} represents the `Delay` from a PCM instance in a QPN (Fig. 4.17). Its queue $q_{id1} = G/G/\infty/IS$ is a delay queue with infinite server scheduling. The actual delay time X is mapped to the service demand of this place $\mu_{id1}(TokenData) = Ph(X)$. As before, $Ph(x)$ is a function transforming the distribution function specified by X into a phase-type distribution. X must not include parameter dependencies (see Chapter 4.2.1).

Entry Level System Call Mapping `EntryLevelSystemCalls` (Fig. 4.18) requires solving parametric dependencies, because the input parameter characterisation of such a call may depend on formerly recorded output parameter characterisations from other `EntryLevelSystemCalls` in the token's variable binding.

The immediate transition t_1 fires whenever a token is available in p_{id1} (i.e., $w_1(t_1) = 1$). It adds a token to both p_{id5} and $p_{id2,id3}$. The place p_{id5} is used to mem-

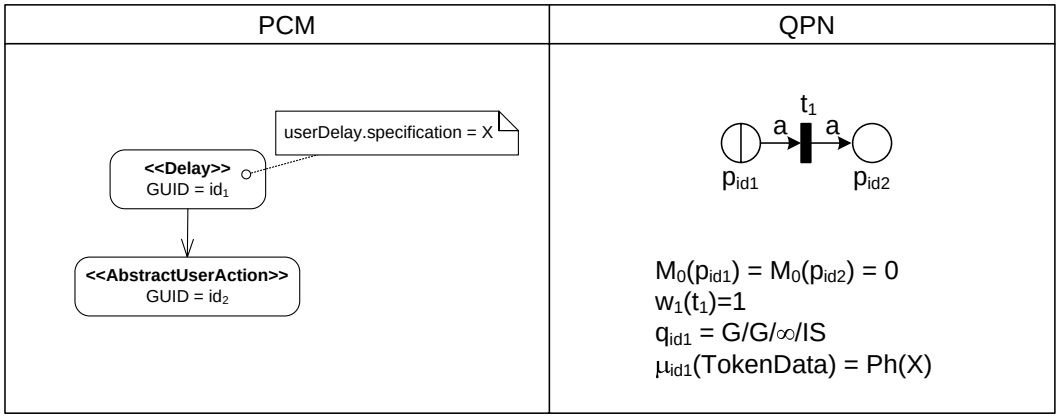


Figure 4.17: Mapping PCM2QPN: Delay

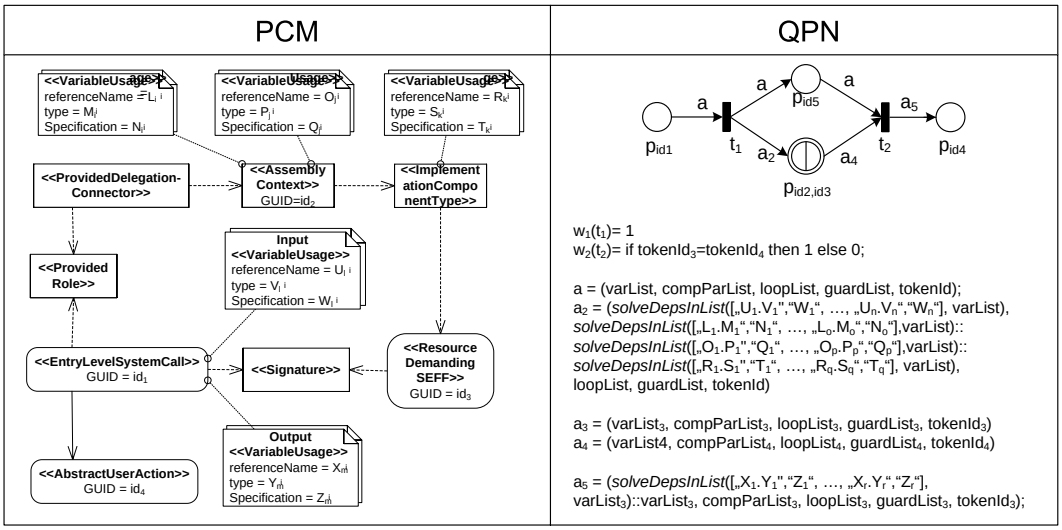


Figure 4.18: Mapping PCM2QPN: EntryLevelSystemCall

orise the users currently executing the called RDSEFF, whereas $p_{id2,id3}$ is a subnet place representing an RDSEFF (id_3) within a specific assembly context (id_2). The original token a gets forwarded to p_{id5}.

a_2 prepares the current token for firing into the RDSEFF subnet place. It adds the input parameter characterisations (reference U_l) and the component parameter characterisations from domain experts (reference L_i), software architects (reference O_j), and component developers (reference R_i) to the token's data. The characterisation can include dependencies to other parameter characterisations, thus a_2 calls the function $solveDepsInList$ (Listing 4.1, in Standard ML [MTH90]), which resolves those dependencies, before adding them to the token's data.

This function simply substitutes variable characterisations from the list $varList$ of formerly recorded output parameter characterisations of other `EntryLevelSystemCalls`. Thus, $solveDepsInList$ tokenises each $varSpec$, which represents a stochastic expression representing the parameter dependency from the PCM instance. This yields a list of strings, for example "a + b" gets tokenised to "a", "+", "b". With this list, $solveDepsInList$ calls the function $solveDeps$ and also forwards the output parameterisation (z).

$solveDeps$ (Listing 4.1) iterates over the list of strings and, upon finding a variable name also present in the passed list of output parameter characterisations, substitutes the current string with the variable specification ($varSpec$) of that characterisation. This ensures that the current bindings of output parameter characterisations are used in the stochastic expressions for the input parameter characterisation of the current `EntryLevelSystemCall` if corresponding parameter dependencies had been specified.

The resulting stochastic expressions do not contain any variable names, but only random variables according to Chapter 3.3 as operands. The actual solution of the resulting stochastic expressions, which may involve arithmetic operations, yields a single random variable useful in the Petri net. The semantics of this solution have already been described in Chapter 3.3.6 and are not given here in Standard ML for brevity.

t_1 fires a token into the subnet place $p_{id2,id3}$, which represents the called `ResourceDemandingSEFF` in a specific `AssemblyContext`. The mapping determines this RDSEFF from the PCM instance (cf. Fig. 4.18) by using the `ProvidedRole` referenced by the `EntryLevelSystemCall`, which is provided by the PCM `System`. With the `ProvidedRole`, the `ProvidedDelegationConnector` and the corresponding `AssemblyContext` can be determined. Its encapsulated com-

ponent contains a set of RDSEFFs, which each reference a signature. Matching the referenced `Signature` from the `EntryLevelSystemCall` then yields the called RDSEFF (Fig. 4.18).

```
1  (* initiates solving parametric dependencies for a list of stochastic expressions *)
2  solveDepsInList : list VarSpec * list VarSpec -> list VarSpec
3      (* termination condition *)
4  fun solveDepsInList (nil,z) = nil
5      (* take the first element from the first list *)
6  |   solveDepsInList ((varName,varSpec)::t, z) =
7      (* solve deps for first element, then continue with rest *)
8      (varName,solveDeps(tokens(varSpec),z))::solveDepsInList(t,z)
9
10 (* substitutes the strings in the list of the first parameter with *)
11 (* variable specifications from the second parameter *)
12 solveDeps : list string * list VarSpec -> string
13     (* termination condition *)
14 fun solveDeps (nil,t) = nil
15     (* could not resolve dependency, use original value *)
16 |   solveDeps (s1::s, nil) =  s1
17 |   solveDeps (s1::s, (varName, varSpec)::t) =
18       (* found dependency *)
19       if compare(s1,varName) = EQUAL
20       (* substitute s1 with the current spec assigned to *)
21       (* the variable with the same name and continue with *)
22       (* the rest of the tokens *)
23       then varSpec^solveDeps(s,(varName,varSpec)::t)
24       (* no dependency found so far, *)
25       (* continue searching for the current varName with *)
26       (* the rest of the VarSpecs, and for the other tokens *)
27       (* again with all VarSpecs *)
28       else solveDeps(s1,t)^solveDeps(s,(varName,varSpec)::t)
```

Listing 4.1: Functions for Solving Parametric Dependencies

The specifications of a_3 and a_4 forward tokens to the transition t_2. This transition fires if a token with a $tokenId$ matching one of the tokens waiting in p_{id5} returns from the RDSEFF subnet (i.e., $w_2(t_2) =$ if $tokenId_3 = tokenId_4$ then 1 else 0;). It then fires a token into place p_{id4}. a_5 adds the specified output variable usages (reference X_i) to the list $varList_3$ of currently valid parameter characterisations again using the $solveDepsInList$ function, because the output variable characterisations may include parameter dependencies. Notice that a_5 also removes the component parameter characterisations by using $compParList_3$ instead of $compParList_4$.

This concludes the mapping to QPN fragments from usage model parts within a PCM instance.

Example As an example for an HQPN resulting from mapping a PCM usage model, Fig. 4.19 shows an excerpt of the HQPN resulting from transforming the

usage model in Chapter 4.2.2. The illustration contains four of the actually seven QPN subnets created by the mapping. The nets resulting from the branch and loop behaviours have been omitted for brevity.

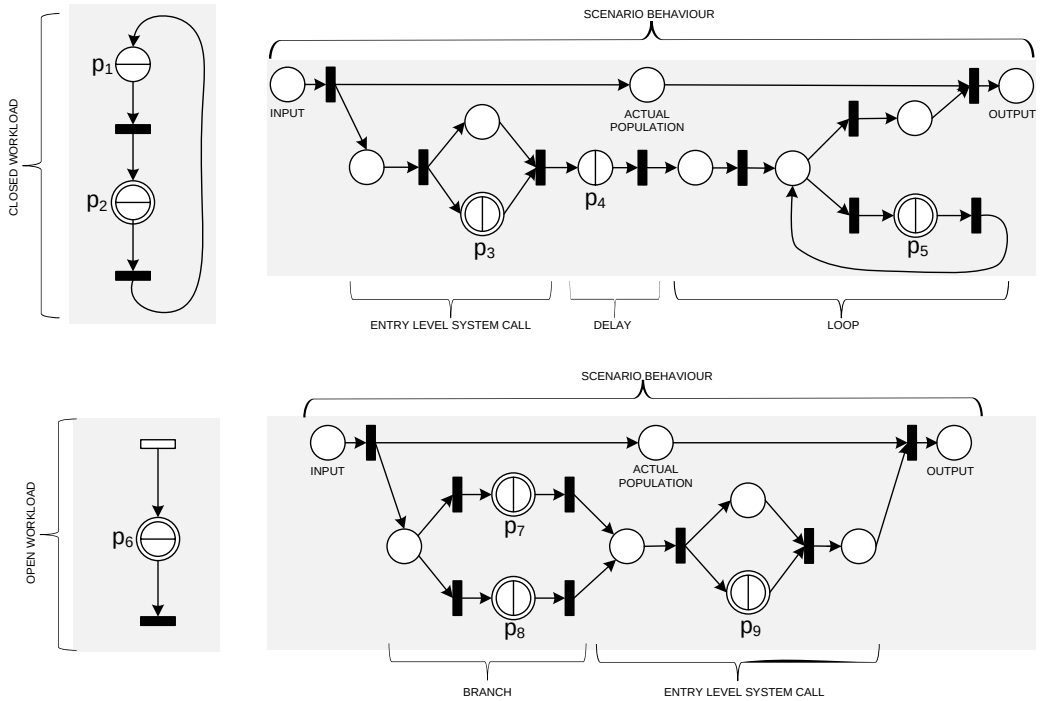


Figure 4.19: QPNs resulting from mapping a PCM Usage Model (Example)

The net contains a QPN subnet for the `ClosedWorkload` in the upper left corner, which (using place p_2) calls the QPN subnet for the corresponding `ScenarioBehaviour` in the upper right corner of the figure. That subnet again calls further subnets not shown here with the places p_3 (RDSEFF) and p_5 (Loop Behaviour). The queueing place p_4 realises the `Delay` from the example.

Furthermore, the net includes a QPN subnet for the usage scenario with the `OpenWorkload` (lower left corner of the figure). With its place p_6, it calls the subnet on the lower right corner of the figure, which represents the `ScenarioBehaviour` of this usage scenario. The subnet-places p_7 and p_8 represent the branched behaviour, and the subnet-place p_9 the RDSEFF called by the `EntryLevelSystemCall` in this `ScenarioBehaviour`.

4.4.3 RDSEFF Semantics

The following describes the mapping from RDSEFF fragments to QPN fragments and subnets. As the mapping is partially similar to the mapping of usage models, the following only describes the mapping rules for action meta-classes, which differ from the usage model mapping. This includes `InternalAction`, `ExternalCallAction`, `SetVariableAction`, `ForkAction`, `AcquireAction`, and `ReleaseAction`. For other action meta-classes (i.e., `ResourceDemandingBehaviour`, `BranchAction`, `LoopAction`, and `CollectionIteratorAction`, the following will briefly sketch their mapping, which is similar to mappings from the usage model meta-classes.

In particular, the mapping of `ResourceDemandingBehaviours` is comparable to the mapping of the usage model's `ScenarioBehaviours`. It results in a QPN subnet with an input, output, and actual population place (as in Fig. 4.14).

Furthermore, the mapping for `BranchActions` and `LoopActions` is similar to the mapping of the usage model's `Branch` and `Loop` respectively (Fig. 4.15 and 4.16). If a `BranchAction` includes `GuardedBranchTransitions`, first the function $solveDepsInList$ needs to be executed on the included branch conditions, then the branch condition is added to the token's $guardList$. By solving the resulting stochastic expression respecting the formerly evaluated branch conditions (see Chapter 4.3.1), the mapping can then determine the branch probabilities. Also the loop iterations specifications in `LoopActions` need to be processed by the $solveDepsInList$ function, because they might include parametric dependencies in RDSEFFs. `CollectionIteratorActions` are mapped equally to `LoopActions`, i.e. the mapping does not respect stochastic dependencies included in their loop bodies as explained in Chapter 4.3.1.

The following describes the mapping for the other RDSEFF action meta-classes, i.e., `InternalAction`, `SetVariableAction`, `ForkAction`, `AcquireAction`, and `ReleaseAction`.

Internal Action An `InternalAction` can contain multiple `ParametricResourceDemands`, which each reference a certain `ProcessingResourceType`. The `ResourceContainer` the component is deployed on needs to have a `ProcessingResourceSpecification` referring to such a `ProcessingResourceType`.

The mapping determines the current RDSEFF's `ResourceContainer` via the `AssemblyContext` and the `AllocationContext` referencing it. As there

is at most one `ProcessingResourceSpecification` referring to a specific `ProcessingResourceType` inside a container, the mapping is unambiguous.

The mapping to QPNs (Fig. 4.20) produces a single queueing place p_{idi} for each `ProcessingResourceSpecification` with the GUID id_i. There is no counterpart for the `ProcessingResourceTypes` in the QPNs, as they are only used to create the link to the `ProcessingResourceSpecifications`.

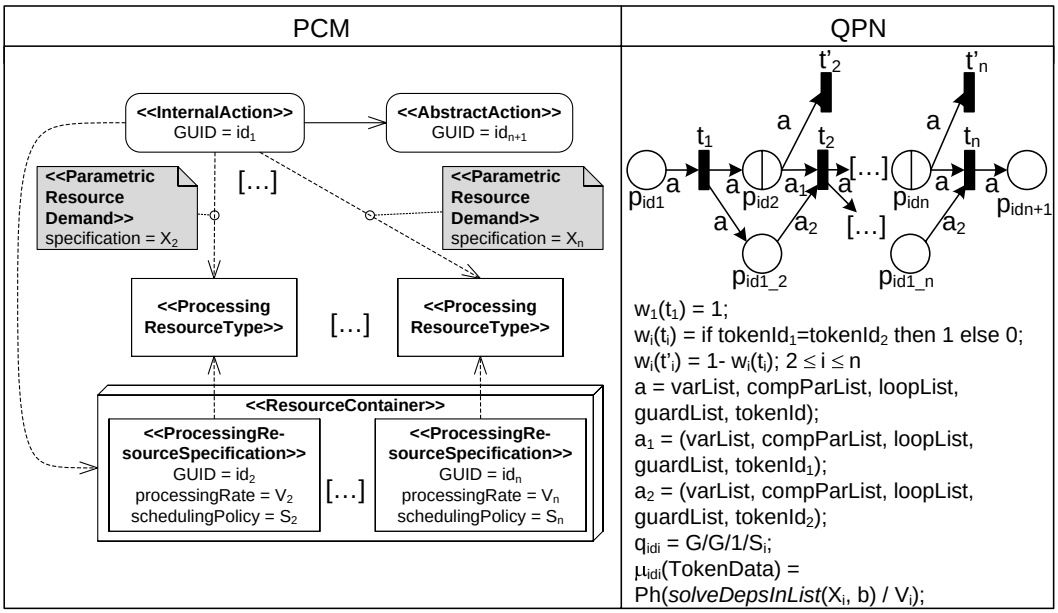


Figure 4.20: Mapping PCM2QPN: InternalAction

The queueing places p_{idi} are of type $G/G/1$ and adopt the same scheduling policy (i.e., First Come First Serve, Processor Sharing, or Infinite Server) as the `ProcessingResourceSpecifications` they are representing. The mapping creates such a queueing place for each `ParametricResourceDemand`, thus in the complete HQPN there are multiple queueing places each representing the same processing resource.

To correctly imitate the behaviour of such a resource, these multiple places for a single processing resource are part of a fusion set. If a transition fires a token into such a place, it is also fired into all other places of the fusion set (i.e., the token is replicated). This ensures that the queues of these places contain copies of the same tokens and the contention delays can be determined correctly respecting all other tokens requesting service from the same resource.

This also implies that the net has to destroy the replicated tokens after the re-

source has served them and only forward the tokens originating from the current RDSEFF. The transitions $t'_2, ..., t'_n$ (cf. Fig. 4.20) are responsible for the destruction of replicated tokens, where n denotes the number of `ProcessingResourceSpecification` the RDSEFF accesses. The net memorises the requests from the RDSEFF to a resource with the places p_{id1_i}. Matching the $tokenIds$ from the memorised tokens and the tokens departing from the queueing place p_{idi} decides whether the net destroys a departing token with the transition t'_i or whether the token gets forwarded using the transition t_i. Thus, all replicated tokens from firing into a fusion set are removed again and cannot alter the behaviour of the current QPN's RDSEFF.

The QPN mapping sequentialises the resource demands specified in the RDSEFF and processes them one after another. If a component service uses resources in parallel, component developers must make this explicit in the RDSEFF using a `ForkAction` and multiple `InternalActions`.

The resource demands in RDSEFFs map to the service demand of the corresponding queueing places, i.e., $\mu_{idi}(TokenData) = Ph(solveDepsInList(X_i, b)/V_i)$, where X_i relates to the `ParametericResourceDemand` and V_i relates to the `processingRate` of the accessed resource. First, the mapping resolves parametric dependencies with the $solveDepsInList$ function. Then, it divides the demand by the processing rate of the `ProcessingResourceSpecification`. This yields the actual time distribution requested by the component service for computing on the resource. As before, the function $Ph(x)$ maps the time distribution to a phase-type distribution. After all resource demands have been processed, a token continues with the place created for the successor action of the `InternalAction`.

External Call Action The mapping for `ExternalCallActions` distinguishes two cases: either the call is directed at another component or the call is directed at a system external service. In the first case, the mapping is almost equal to the mapping of `EntryLevelSystemCalls` (Fig. 4.18). Instead of deriving the called RDSEFF via the system `ProvideRoles` as in the mapping for `EntryLevelSystemCalls`, the mapping follows the `RequiredRoles` of the current component, gets the corresponding `AssemblyConnector`, and takes its providing `AssemblyContext` (also see Chapter 3.2.4). This `AssemblyContext` encapsulates a component, from which the RDSEFF can be determined. As the rest of the mapping is equal to the mapping of `EntryLevelSystemCalls` (cf. Fig. 4.18), a further description is omitted here to avoid redundancy.

In the second case (i.e., the call is directed at a system external service), the mapping for `ExternalCallActions` is different compared to `EntryLevelSystemCalls`. Fig. 4.21 depicts the mapping, which is similar to the mapping for `Delays` from the usage model.

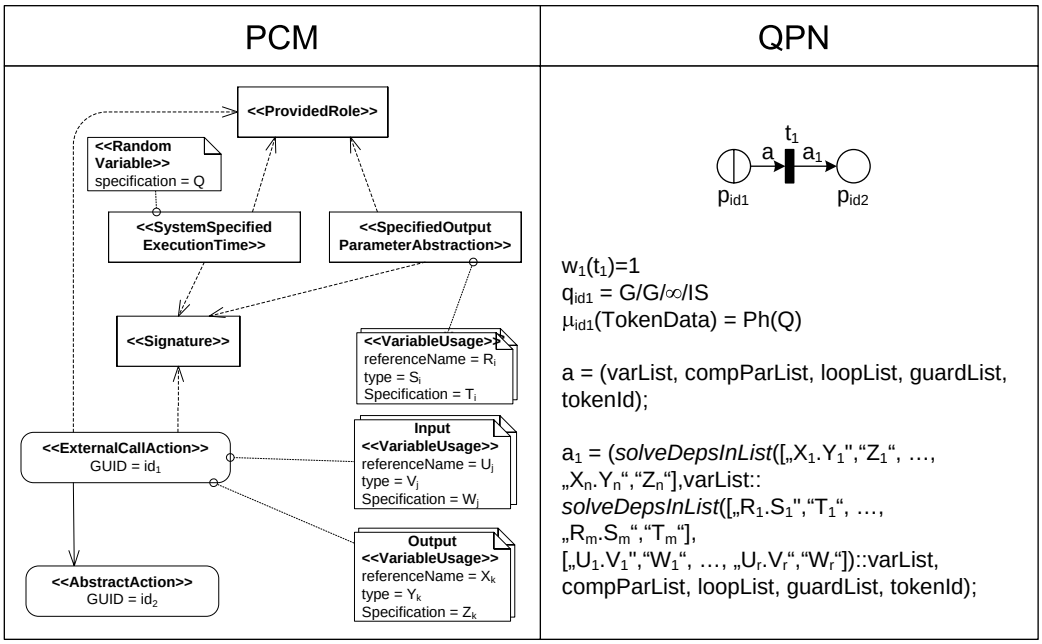


Figure 4.21: Mapping PCM2QPN: ExternalCallAction for system external service

For system external services PCM instances contain no RDSEFFs, but a `SystemSpecifiedExecutionTime` with a random variable specification Q. Using the `ProvidedRole` and `Signature` referenced from the `ExternalCallAction`, the mapping determines the corresponding `SystemSpecifiedExecutionTime`, which also references these model elements. The random variable specification Q is used as a demand for the timed queueing place p_{id1} using the function $\mu_{id1}(TokenData) = Ph(Q)$. $Ph(x)$ determines a phase type distribution for any distribution function x.

Finally, the mapping has to include the `SpecifiedOutputParameterAbstraction` into the token's $varList$. There are two parameter dependencies in this case, which the mapping resolves using the function $solveDepsInList$. First, the `VariableUsages` of the `SpecifiedOutputParameterAbstraction` may depend on the input parameter characterisations of the `ExternalCallAction`. Then, the output parameter characterisations of the `ExternalCallAction` in turn depend on the `SpecifiedOutputParameterAbstractions`.

Set Variable Action These actions specify characterisations of output parameters of a RDSEFF. In the QPN for the RDSEFF, these characterisations must be added to the current token's variable list b. The mapping (Fig. 4.22) recognises that the characterisation may include parameter dependencies. Therefore, it calls function $solveDepsInList$, before adding the output variable characterisations to the current token's variable list.

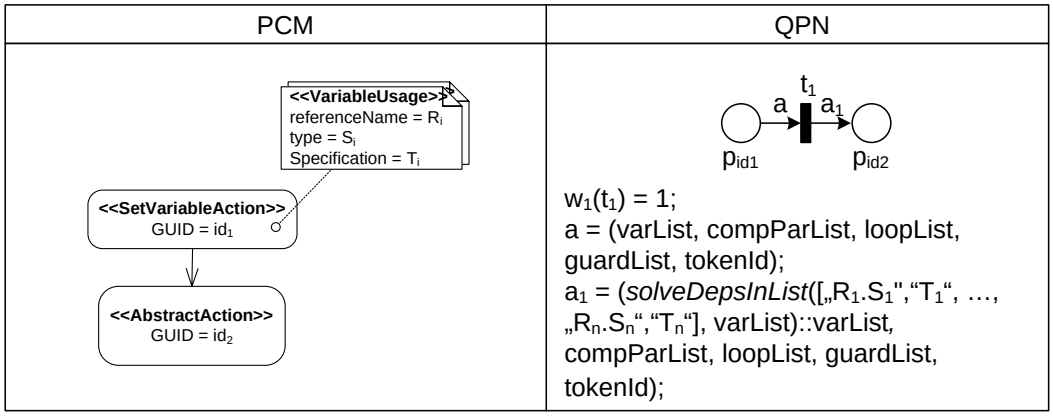


Figure 4.22: Mapping PCM2QPN: SetVariableAction

Fork Action As `ForkActions` are subject to research by Happe [Hap08], the mapping described here (Fig. 4.23) only includes a simple example of an asynchronous fork, where forked behaviours do not join after termination. Happe's thesis details on the semantics for synchronous forks and replication.

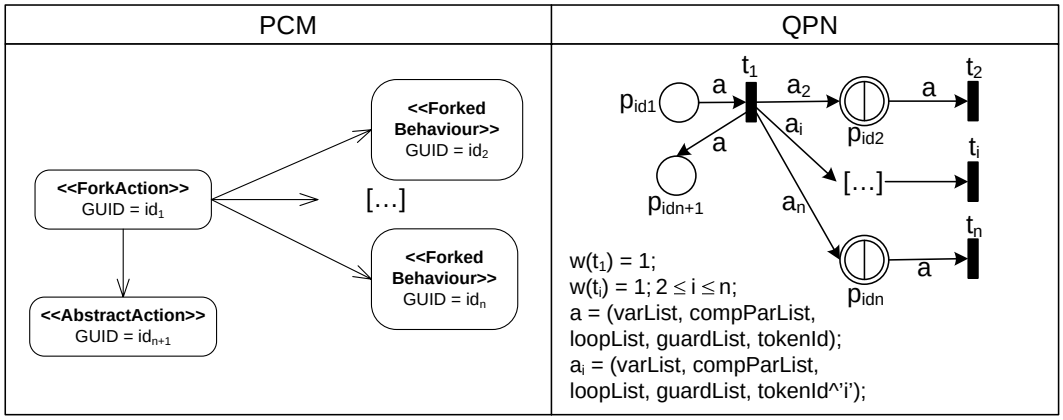


Figure 4.23: Mapping PCM2QPN: Fork Action

The transition t_1 fires a copy of the current token into each QPN subnet place p_{idi} representing the `ForkedBehaviours`. These behaviours are treated equally to `ResourceDemandingBehaviours`. Upon firing t_1, the mapping does not alter the values of the current token's variable except for the user ID h, where it attaches the number i of the forked behaviour. This ensures that the copied tokens have unique user IDs and can correctly use queueing places representing `ProcessingResources` without interfering each other.

After finishing the execution of the `ForkedBehaviours`, i.e., arriving at the output place of the respective QPN subnets, the net destroys the copied tokens using the transitions $t_2, ..., t_n$. Concurrently to the `ForkedBehaviours`, the net continues execution at the successor action of the `ForkAction` represented by the place p_{idn+1}.

Acquire/Release Action `AcquireAction` and `ReleaseAction` of RDSEFFs handle the locking of passive resources, such as semaphores, thread instances, or database connections. The mapping to QPNs produces a dedicated place for each passive resource and initialises it with as many tokens as are specified as its capacity in the PCM instance (Fig. 4.24-4.25). The colour of these tokens is irrelevant, as they need not to be distinguished. Therefore, the colour is called X, and its instances are called x in the following.

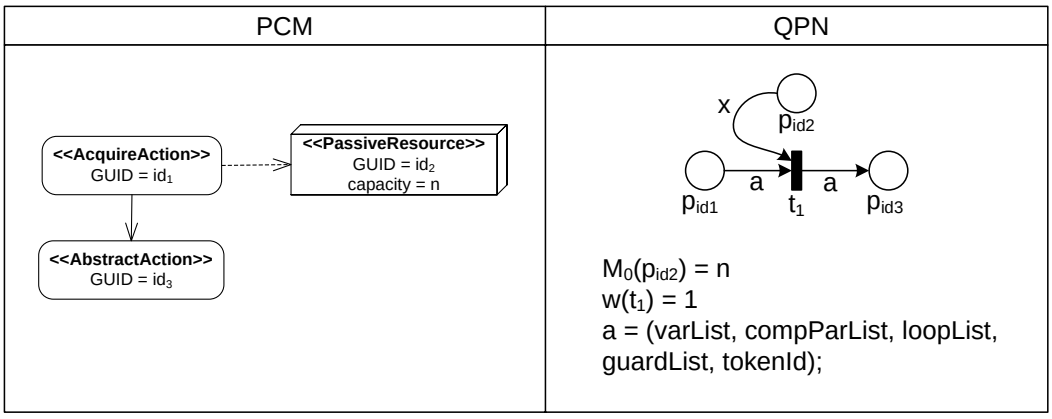


Figure 4.24: Mapping PCM2QPN: AcquireAction

The `AcquireAction` (Fig. 4.24) leads to a QPN that blocks tokens in place p_{id1} until tokens are also available in place p_{id2}, in which case the immediate transition t_1 can fire. It destroys the x-token and forwards the a-token without changes. Afterwards the token continues with the successor action of the `AcquireAction`.

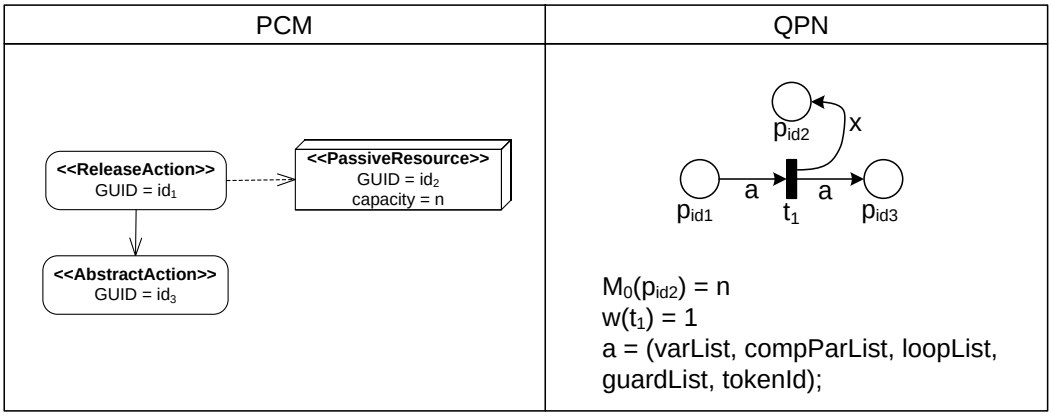


Figure 4.25: Mapping PCM2QPN: ReleaseAction

The `ReleaseAction` (Fig. 4.25) is represented by a QPN, which creates an x-token with the immediate transition t_1 and fires it to the place representing the passive resource p_{id2}. Now other tokens may acquire this instance of the passive resource again.

Example As an example for an HQPN resulting from mapping a PCM RDSEFF, Fig. 4.26 shows an excerpt of the HQPN resulting from transforming the RDSEFF example in Chapter 4.3.2. The figure contains only the QPN subnet representing the topmost `ResourceDemandingBehaviour`, while neglecting the nested loop or branch behaviours.

As any QPN subnet, this subnet contains an input and output place and a place counting the "actual population" (left side of the figure). After being fired from the input place, a token waits until it can acquire a token from place p_1, which represents a passive resource. Then, it executes an `InternalAction` with a resource demand to a CPU, which the QPN represents with the queuing place p_2. The subnet place p_3 links to the subnet representing the RDSEFF called by the `ExternalCallAction` after the `InternalAction`. After completing the call, tokens release the formerly acquire passive resource, therefore the net fires a token into p_1, where it can be acquired by subsequent requests.

The `ReleaseAction` is followed by a `BranchAction` with two branched behaviours, represented by the subnet-places p_4 and p_5. The subnet-place p_6 stands for the loop behaviour executed by the `LoopAction` executed after the `BranchAction`. After executing the loop, the transition after p_7 represents control fork from the RDSEFF. It fires tokens into the subnet-places p_8 and p_9, which have been

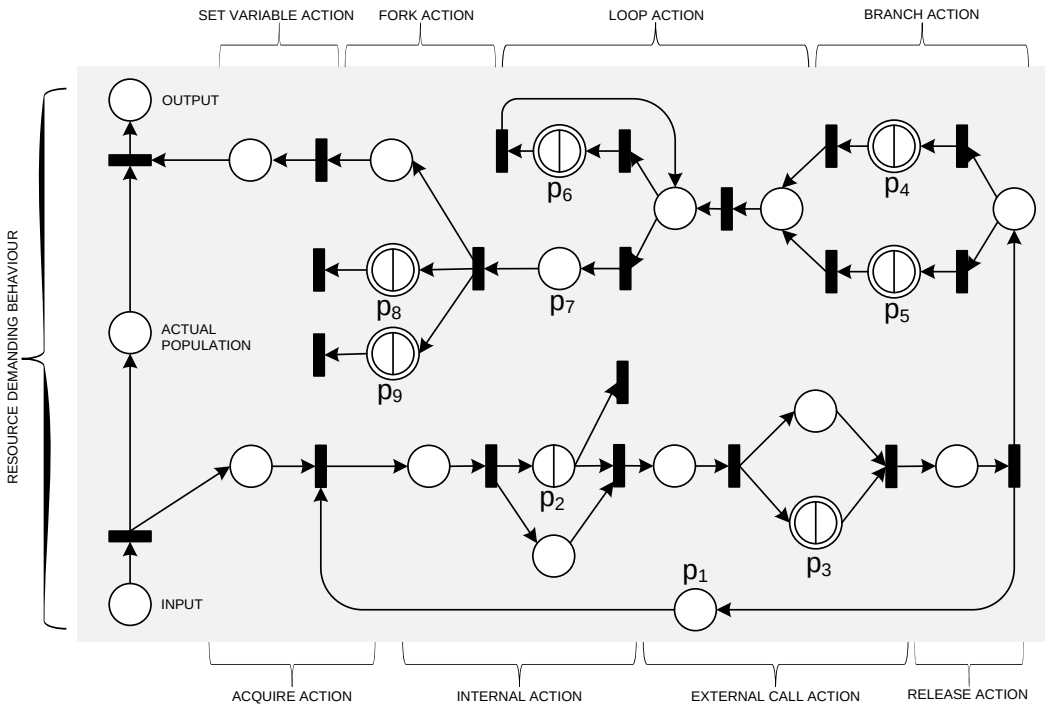


Figure 4.26: QPNs resulting from mapping a PCM RDSEFF (Example)

created by the mapping for the `ForkedBehaviours`. Finally, the second to last transition of the net realises the `SetVariableAction` of the RDSEFF, before a token is fired into the output place and the control flow is returned to the caller of the RDSEFF.

4.4.4 Limitations and Assumptions

The mapping from a PCM instance to a HQPN as described before bears some limitations discussed in the following:

- **Approximation with Phase-Type Distributions:** The mapping assumes that it is possible to accurately approximate the PCM's general distribution functions with phase-type distributions [BH07]. The prediction error introduced by this approximation remains to be quantified.
- **Stochastic Independence:** The mapping assumes stochastic independence between the random variables used in the PCM instance. Other than the semantics of the simulation SimuCom [Bec08], it does not draw samples of the random variables for resource demands or parameter abstractions and propagates

them through the architecture. Instead, the mapping includes the random variables into the data carried by each token. The assumption of stochastic independence allows using analytical methods to solve the model. However, it can lead to inaccuracies if a model contains several of these dependencies (e.g., if a parameter characterisation is used twice to produce a resource demand).

- **Unsupported PCM Features:** The mapping does not support `INOUT` parameters in RDSEFFs, as well as composite components, or incomplete components, such as `ProvidedComponentTypes` and `CompleteComponentTypes`. Including these features remains future work.

4.5 Summary

After the initial description of the PCM in Chapter 3, this section has described extensions to the model, which aim at reflecting the influence of the usage profile on the performance of a component-based system. Chapter 4.1 introduces a new method for modelling input, output, and global parameters. It allows modelling performance relevant aspects of such parameters using random variables and is used both by the PCM usage model and the PCM RDSEFFs. The usage model from Chapter 4.2 allows a description of user behaviour at the system boundaries. Domain experts can model this behaviour with stochastic models, which are able to express the uncertainty during early development stages. For including parameter dependencies into the specification of software components, Chapter 4.3 has introduced the RDSEFF language, which allows such dependencies for branch probabilities, loop iteration number, resource demands, and parameters passed to required services of a component. If all components in an architecture are modelled with this language, the usage profile of each individual component can be determined automatically. Chapter 4.4 complemented the former informal description of the modelling languages with a mapping to HQPNs to capture the performance-related behaviour of PCM instances formally.

Chapter 5

Generating RDSEFFs from Java Code

5.1 Motivation

The PCM RDSEFFs language introduced in Chapter 4.3 allows specifying the performance properties of a software component's service in a parametrised way.

For *new* component services, which are planned for a new architecture but not yet realised, component developers have to create the RDSEFF specification manually. They have to model the expected control flow for the RDSEFF and estimate resource demands as well parameter dependencies on branches, loops, and external service calls. Estimations can result from experience with similar systems, measurement of prototypes, or simply guesswork. Smith et al. [Smi02] provide many hints for obtaining useful estimations for such models.

For *existing* component services, which have already been implemented in code, generating RDSEFFs at least partially appears possible. Tools can retrieve information needed for an RDSEFF from existing development artifacts, such as design documents, source code, or already conducted performance measurements. Furthermore, tools can execute existing software components and measure their performance properties. In the case of existing software components, the PCM supports analysing changing usage profiles and answering sizing questions.

Automated creation of performance specification has several potential benefits over manual RDSEFF specifications for existing code. Automatic generation of RDSEFFs is less time-consuming than manual specification, especially for large software components. The resulting models are less error-prone, because of the missing human influence, as shown later in a case study. Furthermore, they can be more accurate, because the included information does not rely on uncertain estimations,

but on actual measurements. Giving component developers tools to automatically create performance models from their components could also overcome their inhibitions about performance modelling because of the expected high effort. Thereby, model-based performance analysis could become more prevalent.

Notice, that it is not sufficient to execute a component service in a specific hardware environment, with specific input parameters, and specific external services and measure the time it spends on each resource to create a parametrised RDSEFF. The RDSEFF is intended for reuse in different environments and is parameterised over all external influence factors. Therefore it requires executing a component service over a range of input parameters to reveal the parametric dependencies between the inputs, resource demands, external service calls, and component behaviour. While it is effortless to create non-parametrised RDSEFFs of component services for a specific environment by simple execution and measurement, the resulting specification would not be suitable for third-party reuse as it would not contain parametric dependencies.

There are several possible approaches for automatic generation of performance models for existing components, ranging from static code analysis and dynamic program analysis to symbolic execution. Chapter 5.2 provides an overview of these approaches and discusses related work. For PCM RDSEFFs, a hybrid reverse engineering approach involving static code analysis and dynamic program analysis has been proposed [KKKR08]. Chapter 5.3 sketches its process model and provides pointers to already completed work for this process.

In the scope of this work, a static code analysis has been implemented as an Eclipse plug-in called Java2PCM [Kap07]. It is capable to derive initial RDSEFF structures from arbitrary Java code. Chapter 5.4 describes the mappings from Java code to RDSEFFs supported by Java2PCM, before Chapter 5.5 provides an overview on its implementation. The application of Java2PCM on a larger component-based software architecture in Chapter 5.6 evaluates its correctness and proposed benefits.

5.2 Techniques for Automatic Performance Model Generation

This section lists several techniques useful for automatic performance model generation and discusses their benefits and drawbacks.

Design Model Analysis For existing software components, design documents (e.g., UML diagrams) are often available, which could be exploited for the generation of performance models. In fact, many approaches targeting the generation of formal performance models (such as queueing networks, stochastic process algebras, stochastic petri nets, etc.) from UML diagrams have been proposed [BDIS04, CDI01].

These approaches require performance-related annotations (e.g., resource demands, loop iteration numbers, branch probabilities) on the often purely functional models to perform their transformations into performance models. Woodside et al. call these additional annotations "completions" [WPS02]. The UML SPT profile [Obj05b] or the UML MARTE profile [Obj07a] offer UML extensions to specify such annotations.

Usually, manual specification of these annotations involving estimations [Smi02] is assumed, because these approaches target early life-cycle performance analysis, where no source or binary code is available that could provide the needed information. Therefore, model-based performance prediction methods do not offer support for automatic generation of performance annotations based on measurements, but require performance modelers to include measured data into annotations by hand.

In general, using existing functional design models for automatic performance model generation is limited, because of the missing required annotations. Only control flow structures from such models are useful for performance models. Another drawback of using design models are potential inconsistencies between design and source code, which leads to inaccurate performance models.

Static Code Analysis If an existing software component is available as source code (or byte code), static code analysis is possible. It can involve parsing the source code and operating on its abstract syntax tree to derive information needed for performance models. As for design model analysis, source code analysis provides control flow structures, however without inconsistencies between design and implementation. The resulting control flow model is complete, whereas dynamic program analysis (black box testing) might not find certain branches, which are not reached because of the chosen input parameters.

Additionally, constant (i.e., fixed in code) loop iteration numbers and constant values for parameters of external service calls can be determined via static source code analysis. In simple cases, the analysis can also recognise dependencies between input parameters and control flow guards, loop iterations, or external service

calls by tracing parameters through code. It is however limited for more difficult dependencies involving complicated computations or polymorphism. In general, the halting problem underlies static code analysis, and the iteration numbers of arbitrary loops and the values of parameters are not determinable with this method.

Resource demands are in general not determinable via static code analysis as they require the execution of code. It is however possible to determine *that* certain resources are used by code, for example by identifying API calls directed at resources such as hard disks or network devices.

In the area of reverse engineering [Kos05], there are many approaches and commercial tools (IBM RSA, Borland Together) deriving UML models from source code. However, these methods and tools mainly focus on the reconstruction of functional models and do not provide advanced support for deriving performance related information. Furthermore, these tools do not abstract from the control flow of a software component as RDSEFFs do by combining multiple statements into a single internal action. Control flow models resulting from the static analysis of current reverse engineering approaches would be too fine-granular for a performance analysis.

In the area of performance engineering, no known approach utilises static code analysis for the generation of parametrised performance models. Many performance prediction methods only support models for a specific context and do not aim at reusable models, which require parameterisation. For such throw-away models, executing the code with specific input parameters on a specific machine to determine performance properties is easier and more accurate than static code analysis.

Program Slicing A special form of static code analysis is program slicing [Wei81, Tip94]. A program slice is a code excerpt, which potentially affects the values of a variable at a specific location in a program. The location or point of interest is called slicing criterion, which is usually specified in combination with the program's variables, which are of interest. There is a distinction between static slicing, which does not make assumptions on input parameters, and dynamic slicing, which is performed for a given test case.

Originally, Weiser designed program slicing for debugging [Wei81], because if an incorrect value of a variable is detected at some point of the program, the corresponding bug is likely located within the program slice for this variable. Program slicing has also been used for parallelisation, program differencing and integration, software maintenance, testing, reverse engineering, and compiler tuning [Tip94].

In the context of automatic performance model generation, program slicing is potentially useful to determine parameter dependencies between, for example, control flow guards using local variable names and input parameters. The program slice would be the basis to define the parameter dependency or derive some abstraction of it to use in RDSEFFs, if it is too complex. However, this work does not investigate the use of existing program slicing approaches for this task and regards it as future work.

Dynamic Program Analysis Dynamic program analysis includes executing an existing software component and measuring its resource demands and the amount of external service calls. It is also called performance analysis or profiling. Component execution requires the availability of all required services. Therefore, either actual components implementing the required services must be available, or some kind of test-bed needs to be created, which intercepts call to these services and simulates their behaviour.

Dynamic analysis has several benefits. It can potentially determine all information needed for a performance model including control flow structure, resource demands, and parameter dependencies. Furthermore, dynamic analysis is possible even if no component source code is available, therefore it is the only method supporting the analysis of black box components.

As a drawback, it is not trivial to generate a parametrised performance model (such as an RDSEFF) via dynamic analysis. To parametrise the specification for different usage profiles, it involves testing component services for their whole input parameter space, which is infeasible in general. To parameterise for different hardware resources, it requires testing on different systems. To parameterise for different external services, it involves testing the execution over the whole output domain of external services.

Woodside et al. [WVCB01] have proposed a dynamic analysis approach to determine resource demands of software components parametrised for different usage profiles, which they call ”resource functions”. This approach includes executing a component service numerous times with varying input parameters while measuring execution times. They use statistical methods such as linear regression and regression splines to create the resource functions from the measured execution times. However, the resource functions in this approach are not parametrised over different hardware and require a reference platform. Furthermore, the approach does not consider output parameters of external services influencing component behaviour.

Meyerhöfer et al. [MN04, MV05] provide a testbed for measuring the execution times and memory consumptions of EJBs. They furthermore sketch a method of measuring resource demands independent from a specific platform [ML05], which involves partitioning the resource demand of a component service into several segments and benchmarking each segment on each platform of interest to be able to derive the actual resource demand. However, their resource demands are not parametrised for different usage profiles.

Symbolic Execution A special form of dynamic program analysis is symbolic execution [Kin76]. It executes existing software components using variables (called "symbols") as input parameters instead of concrete values. Upon reaching control flow branches with guards over the input parameters, symbolic execution uses the guards to produce a formula over the input parameters. Solving such formulas can determine whether specific code statements can be reached at all to assess program correctness.

Although mainly used in the context of model checking and program verification to recognise deadlocks or dead code blocks, symbolic execution could also be used to determine the parameter dependencies on branches, loops, and external service calls for RDSEFFs. It would also be useful to determine the full control flow structure of a component, because the techniques is able to achieve a full path coverage of given code. As symbolic execution with variables as inputs complicates the processing overhead when executing a component, it is generally not suited to determine accurate resource demands with this method.

A drawback of symbolic execution is its scalability, as it is limited when analysing large components. In the context of NASA's Java Pathfinder project, Visser et al. [VHBP00, KPV03] have reported symbolic execution on components with up to 10000 lines of code. To handle more complex components, Sen et al. [SMA05] have proposed combining symbolic execution with generating concrete random values ("concolic testing") to reduce the complexity of the formulas resulting from symbolic execution.

In general, symbolic execution is successful when working with integer or real number input parameters, but it is limited if code statements involve other data types or even polymorphism, because the resulting formulas cannot be solved easily then.

Parizek et al. [PPK06] have used Java Pathfinder to analyse whether given Java code complies to a behaviour protocol specification, which expresses component

service behaviour similarly to RDSEFFs, but does not contain parameter dependencies. The approach could determine violations of the protocols, if the code reached a service call not present in the protocol specification.

Prototype Testing Several researchers have investigated prototype construction and measuring to rapidly generate accurate performance models during early development stages, if no implementation is available.

Hrischuk et al. [HRW95] propose using a prototype, which includes the control flow logic (i.e., the interaction of objects) of an object-oriented system, while neglecting other computations inside objects. Execution of the prototype for different usage scenarios yields different program traces, which include the number of statements executed and their resource demands. This measurement data is the basis for constructing an annotated control flow graph, which is then transformed into a Layered Queueing Network. The LQN needs to be completed for missing parameters, such as the hardware specification. It is doubtful, if such an approach would be useful to construct RDSEFFs, because it involves the execution of pre-defined scenarios, whereas parametrised RDSEFFs need executions representable for all possible usage scenarios of a component service.

Denaro et al. [DPE04] target modelling the performance of J2EE systems. Their main assumption is that the code of individual components has little impact on the overall performance of a distributed application, because the main processing overhead shall result from the middleware. They select performance-critical usage scenarios from design documents and then generate stubs for each involved component. Using workload generators and generated persistent data, these stubs can be executed after being deployed on the target J2EE platform. They directly use the measurement data from this setting to determine the performance properties of a proposed architectural design without construction of a specific performance model. They claim that their method is advantageous over pure middleware benchmarking, because it additionally includes application logic. This method may work for simple components with a business logic not critical for performance, but fails to include the processing overhead of component with more complex calculations.

The approach by Liu et al. [LFG05] combines manual application models with benchmark results from measuring a J2EE platform. The authors make the same assumption as Denaro et al. and emphasise the performance influence by component containers. They use design documents such as activity diagrams to determine the amount container functions (e.g., create a JavaBean, store a JavaBean) processed

by a given design and then use a generic benchmark to determine the execution times of theses functions for a given J2EE container. The data is input for a simple queueing network model of the container infrastructure, which can then be used for capacity planning. This approach is interesting for RDSEFF generation because it targets container independent resource demands for the application models, which are similar to the platform independent resource demands in RDSEFFs.

5.3 A Hybrid Approach for Reverse Engineering RDSEFFs from Code

The reverse engineering process proposed for generating RDSEFFs from Java source code combines static and dynamic analysis in a hybrid approach [KKKR08]. It tries to leverage the benefits of both approaches, i.e., using accurate control flow structures and initial parameter dependencies from static code analysis and resource demands from dynamic analysis. Using a platform-independent abstraction, such as the number of byte code instructions needed, the resource demands shall additionally be parametrised over the underlying resource environment. The process model depicted in Fig. 5.1 is generic and could be used with different source code types, tools, performance models, and performance analysis or simulation methods.

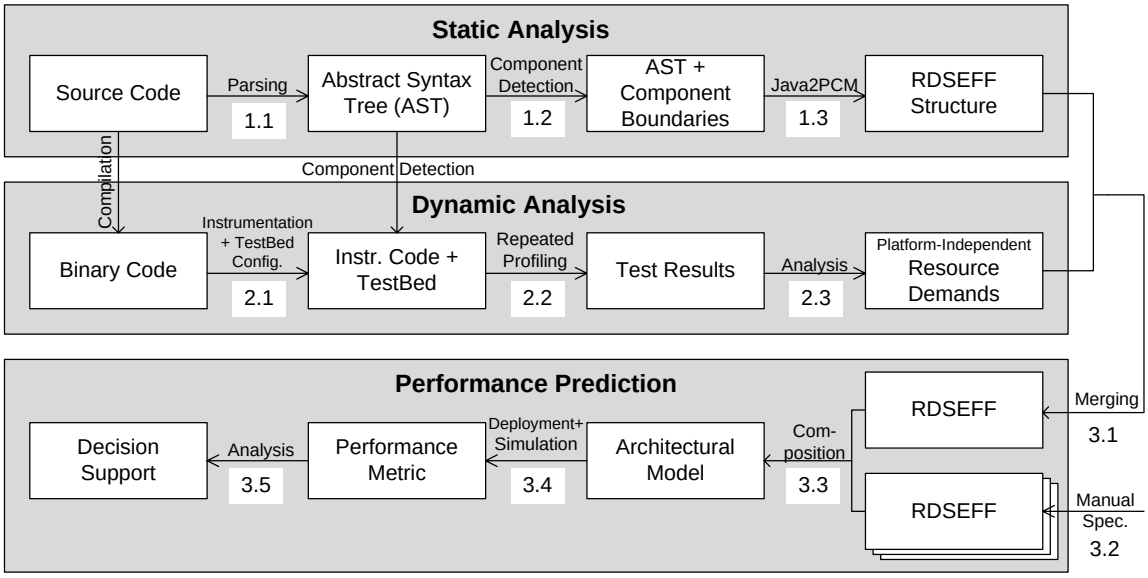


Figure 5.1: Reverse Engineering and Performance Prediction

The process in Fig. 5.1 starts with *static analysis*, which first parses given code to an abstract syntax tree (1.1). Then reverse engineering must identify component boundaries (1.2) in this representation of potentially a whole architecture. This is challenging if the code under study is written in an object-oriented language such as Java, which per se does not support the component paradigm. For this step of design recovery, there exists a large body of research [Kos05]. For identifying PCM components, Chouambe [Cho07] has implemented a component detection tool for arbitrary Java code, which uses different code coupling metrics to identify components among classes.

The component boundaries resulting from the previous step are input to another static code analysis, which produces initial RDSEFF structures with statically determinable performance annotations (1.3). This code analysis was implemented in a tool called Java2PCM for the scope of this thesis (also see the diploma thesis from Kappler [Kap07]). Chapter 5.4-5.6 will described it in more detail.

In parallel to static analysis, the proposed process also performs *dynamic analysis* on existing software components (Fig.5.1). Upon the time of writing, there is still no implementation of this step into tools, therefore the following sketches the planned approach. Dynamic analysis instruments the code with measurement probes and sets up a test-bed to execute the component under analysis (2.1). This involves generating dummy-components for external services, which are not available as code, and defining a set of test cases with representative parameter values for the following execution. With the completed test environment, repeated profiling for all test cases produces measurement data for resource demands and external service calls, which needs to be stored for later analysis (2.2). The overhead for measuring must be excluded from the data.

Resource demands should not be measured as timing values, but as some abstract measure (such as CPU cycles, byte code instruction, or generic work units), to produce resource demand parameterisable for different platforms. Kuperberg et al. investigate the possibilities for platform-independent resource demands [KB07]. The analysis of the measurement results from component execution (2.3) targets deriving dependencies between control flow guards, the amount of external services calls, resource demands and input parameter values. In many cases, linear regression is sufficient to derive a function for resource demands over input parameters from the measured data (see for example [KBH07]). Woodside et al. [WVCB01] used regression splines to determine more complex dependencies in multiple recorded execution times. Krogmann [Kro07] proposes using genetic algorithms on the data

to identify even more complicated patterns.

To complete the reverse engineering process for RDSEFFs, tools shall merge the RDSEFF structure from static analysis with the platform-independent resource demands and additional parameter dependencies from dynamic analysis to get full RDSEFFs (3.1). Afterwards, component developers can put these RDSEFFs into public repositories, where software architects can retrieve them and use them for composition with other (possibly manually specified) RDSEFFs (3.2) to model a component-based software architecture (3.3). With a full PCM instance, QoS analysts may use the analysis and simulation techniques from [BKR07] and Chapter 6 to conduct performance predictions (3.4) and derive decision support (3.5).

5.4 Static Analysis: Mapping Java Code to RDSEFFs

This section describes the second static code analysis step from the previous subsection, which has been implemented as an Eclipse plug-in called Java2PCM [Kap07]. It will first describe how Java2PCM performs a static code analysis using the Eclipse framework and explain the results of the analysis. Afterwards, it explains the different concepts of the transformation from Java code to PCM instances in detail. Finally, the following two subsections briefly discuss its implementation and present a case study of applying Java2PCM to a larger component-based architecture.

Once installed in Eclipse, Java2PCM offers a new context-menu entry "Generate RDSEFF" when right-clicking a Java package, class, or method in the IDE. After choosing this menu entry, Java2PCM uses the Eclipse Java Development Tools (JDT) to parse the selected code and create an abstract syntax tree (AST). It then uses an AST visitor class from JDT, which was extended to produce RDSEFFs elements from Java code fragments. For each element found in the AST, it executes a specific transformation trying to derive as much information for the RDSEFF from code as possible, which will be detailed in the following subsections.

Java2PCM accesses the factories from the PCM's EMF implementation to produce RDSEFF elements and adds them to an object tree representing a PCM `Repository`. After traversing the whole AST, a serialisation of the object tree to an XML document stores the `Repository` to a file. As the included RDSEFFs still lack resource demands and might contain invalid parameter dependencies, component developers have to correct and complete them manually via the PCM-bench editors. Software architects can then use completely edited file for adding the included RDSEFFs into their architectural models.

It is not trivial to map Java code to RDSEFF instances, because each language describes the behaviour of a component on a different abstraction level. The RDSEFF's abstraction level is substantially higher than Java code and focusses on external service calls and resource demands. In contrast, Java code does not provide external service calls as classes do not contain explicit required interfaces, and also does not make all resource usages explicit.

The following describes the tasks performed by Java2PCM: location of external service calls (Chapter 5.4.1), location of resource usage (Chapter 5.4.2), control flow transformation (Chapter 5.4.3), abstraction-raising transformations (Chapter 5.4.4), and establishing parameter dependencies (Chapter 5.4.5).

5.4.1 Location of External Service Calls

When selecting a Java package, class, or method for generating an RDSEFF, the component boundaries are not clear. Java classes do not have explicit provided and required interfaces, although the set of public methods could be interpreted as the provided interface and the set of import statements as required interfaces. While only public methods can be invoked from the outside, not all of them might qualify as provided services, which complicates the identification of the provided interface. Import statements of classes specify which classes or packages outside the current class are required, but they do not distinguish between API calls, component internal, and component external calls, and are therefore not suited as required interface.

In the future, Java2PCM shall use a dedicated component detection approach [Cho07], which provides component boundaries as additional input to Java2PCM besides the selected code. However, its implementation and integration is still missing. In the current implementation, Java2PCM makes default assumptions on the component boundaries explained in the following and lets the user configure them further.

To identify provided interfaces, Java2PCM provides two alternatives. In the first alternative, which is the default setting, Java2PCM simply interprets every public method of a class as a provided service and includes it into a single provided interface for the class, which is interpreted as a component. In the second alternative, which is a configurable option, Java2PCM interprets only methods implementing services of explicitly declared interfaces as provides services and generates PCM provided interfaces for each implemented Java interface.

To identify required interfaces, Java2PCM per default regards every method call,

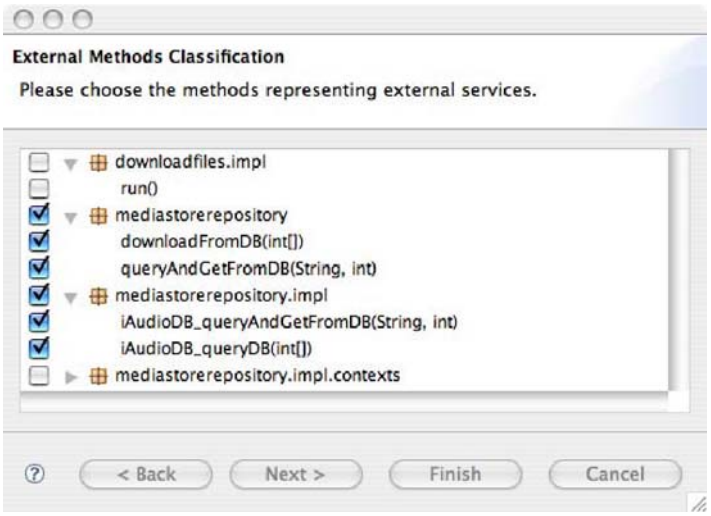


Figure 5.2: Java2PCM: Classifying External Service Calls

which is directed outside of the current package and not directed to the Java API, as a call to an external service. It generates a single required interface in this case and adds all found external service calls as required services. It is furthermore possible to also remove methods calls to sub-packages of the current package from the list of external service calls via a configuration option and treat them as component internal method calls.

As another option, after invocation, Java2PCM initially traverses the AST of the parsed selected code and assembles a list of all included method invocations. It then provides the user a selection dialog depicting the method invocations as a fold-out tree grouped by packages (Fig. 5.2). The user can select complete packages or individual methods calls and classify them as external. Java2PCM then treats all unselected methods as component internal and does not create `ExternalCallActions` for them.

5.4.2 Location of Resource Usages

In RDSEFFs, component developers attach resource usages to `InternalActions`. These reference a resource type and specify the resource demand as a random variable. In principle, RDSEFFs allow the specification of resource demands on different abstraction levels. For example, for each statement of Java code a single `InternalAction` with a unique resource demand could be created. However, this is not desirable, because it would make the model too complex, which might lead to its

intractability when using analysis or simulation tools.

Java2PCM tries to create RDSEFFs with a higher abstraction level by combining all statements between two identified external service calls into single `InternalActions` and creating a combined CPU demand with an amount of zero for them per default. The actual demand (e.g., 1000 CPU cycles) is hardly determinable via static code analysis and must be supplemented manually or by dynamic program analysis. While each statement in Java code potentially also uses main memory, Java2PCM ignores memory usage as the PCM so far does not support analysis facilities for it.

Besides CPU and memory usage, static code analysis can determine that code accesses *other resources* (e.g., storage devices, networks, or passive resources). Therefore, it can create additional resource demands (with zero amount) referencing other resource types than the CPU to make the resulting RDSEFFs more accurate. Especially accessing comparably slow storage and network devices changes the performance of a component service significantly and needs to be included in an accurate performance model.

In Java code, developers usually specify accesses to other resources by using the Java API. For example, they use classes and methods from the package `java.io`. A problem for the static code analysis is that these packages abstract from concrete underlying resources. The data source and destination of an I/O stream is polymorphic and can only be determined at run time. It can for example be a file on a hard disk, a file in main memory, or a network socket. Therefore, Java2PCM creates an abstract I/O resource demand for method invocations directed at the `java.io`-package and leaves it to the component developer to specify concrete resources. Besides the `java.io`-package, Java2PCM also creates I/O resource demands for calls to `java.sql`, `java.util.logging`, `java.util.zip`, `java.imageio`, and `javax.sql`, which use `java.io` internally.

Accesses to *network devices* are a special case for the static code analysis creating RDSEFFs. While they can be identified in code via calls to the `java.net`-package, the PCM does not allow components to access network devices *internally* (RDSEFFs may not reference `CommunicationResourceTypes`). As system deployers can allocate each component only to a single `ResourceContainer`, components must communicate with other components in different resource containers via external service calls. Then, the PCM analysis and simulation tools automatically produce load onto the network device connecting both resource containers. They calculate the byte size of the load by summing up the byte size of the parameter characterisa-

tions passed to the external service call. As the analysis and simulation tools handle this internally, it is not reflected in the RDSEFF specification. Therefore, Java2PCM ignores network accesses in Java code.

So far, Java2PCM does not support *passive resource* usage. An extension into this direction is future work. In fact, the Java API makes the use of many passive resources explicit, as there are dedicated classes for thread pools, semaphores, and monitors. Java2PCM could use calls to these classes in the code to create passive resources as well as acquire/release actions.

5.4.3 Control Flow Transformation

Java2PCM transforms Java control flow statements into RDSEFF control flow elements. Namely, it supports branches, loops, and forks. Notice, that this transformation only occurs, if the Java control flow elements include either external service calls or resource demands to devices other than the CPU in their enclosed code blocks. Otherwise, Java2PCM ignores Java control flow statements to create a higher abstraction level in the resulting RDSEFFs.

`if/else` and `switch/case` statements specify control flow *branches* in Java code. Java2PCM maps `if/else` statements to RDSEFF `BranchActions` and uses the Java boolean expressions to create `GuardedBranchTransitions` with corresponding guards. It furthermore traces the local parameters included in boolean expression back to input parameters (details in Chapter 5.4.5). As the guards in `GuardedBranchTransitions` need to be mutually exclusive, Java2PCM adds the negated version of each previous boolean expression to the successive guards in RDSEFFs.

Java2PCM maps the `case` statements enclosed by a `switch` statement to individual `GuardedBranchTransitions` with `ResourceDemandingBehaviours` for the statements after each `case` statement. Java allows `case`-blocks, which not end with a `break` statement and then also executes the code of the following `case`-block until it reaches a `break` statement. Java2PCM recognises these situations and duplicates the statements inside the following `case`-block into the previous `case`-block and creates the corresponding actions for them.

Java2PCM maps `for` and `while` statements to RDSEFF `LoopActions`. While the number of iterations of Java *loops* are defined via boolean expressions for aborting, RDSEFF loops directly specify the number of iterations as random variables, which might include input parameter characterisations. Therefore, Java2PCM tries

to determine the number of loop iterations from the code. It recognises simple cases, such as statements like `for(i=0;i<X;i++)`, where it uses `X.VALUE` as the number of iterations. Furthermore, it recognises iterations over collection, as it checks whether the loop expression in the code contains an `Iterator` object, then derives the corresponding collection `Y`, and then uses `Y.NUMBER_OF_ELEMENTS` as the number of iterations. Java2PCM also deals with Java loops containing an external service call inside their loop expressions, and adds an `ExternalCallAction` to the loop body behaviour.

However, in general Java2PCM cannot determine the number of loop iterations for arbitrary Java loops as the halting problem underlies static code analysis. If, for example, the loop expression is manipulated inside the loop body, it is usually impossible to derive the number of iterations or the resulting changes to variables statically. If none of the methods for detecting the number of iterations from above is applicable, Java2PCM at least traces the variables inside Java loop expressions back to input parameters (Chapter 5.4.5) and includes the result into the specification of the RDSEFF loop iteration number as a comment. This may give component developers, who complete the RDSEFF specification manually, a hint on how to estimate the actual number of iterations. For the future, dynamic program analysis additionally shall assist in determining the number of iterations.

For control flow *forks* (i.e., thread invocations), Java2PCM detects the use of objects of classes derived from `java.lang.Thread` or classes implementing the interface `java.lang.Runnable`. It maps the `Thread.start()` statement to an RDSEFF `ForkAction`. Afterwards, it uses the statements inside the overwritten method `run()` of the object to create the RDSEFF `ForkedBehaviour`.

5.4.4 Abstraction-Raising Transformation

Besides the combination of multiple statements into single `InternalActions` and the omission of control flow statements, which do not include nested external service calls, as described above, Java2PCM performs several additional transformations, which aim at raising the resulting RDSEFF's level of abstraction.

Java2PCM tries to combine nested branch and loop statements in direct succession. This for example occurs if an `if`-statement is directly nested inside another `if`-statement without any external service calls in between. Java2PCM recognises such situation and combines the included boolean expressions. For example, Java2PCM transforms a statement `if(X){if(Y){}}` into a single `BranchAction`

with a `GuardedBranchTransition` containing the guard specification `X AND Y`. The same is possible for nested loops. For example, if Java2PCM finds a statement like `for(X){for(Y){}}`, where `X` and `Y` are the recognised loop iteration numbers, it creates a single `LoopAction` with `X*Y` as the specified number of iterations.

Furthermore, Java2PCM supports *in-line expansion* of local methods for RDSEFFs. While Java supports structuring large segments of code into different local methods (i.e., methods often classified as `private`), RDSEFFs do not provide such substructures. The use of local methods is an implementation detail aiming at reuse and maintainability, but does not change the performance properties of the code. Therefore, their representation inside RDSEFFs is not necessary. Whenever Java2PCM finds a local method call it continues analysis at the local method and attaches external service calls and important resource accesses inside it to the previously created RDSEFF structure.

```
1  public void printList(List<String> paramElements) {
2      printHelper(paramElements);
3  }
4
5  private void printHelper(List<String> elements) {
6      List<String> listCopy = elements;
7      for (String curElem : listCopy) {
8          System.out.println(curElem);
9      }
10 }
```

(a) Before

```
1  public void printList(List<String> paramElements) {
2      List<String> listCopy = paramElements;
3      for (String curElem : listCopy) {
4          System.out.println(curElem);
5      }
6  }
```

(b) After

Figure 5.3: Java2PCM: Method Inlining

Fig. 5.3 provides an example for method inlining. The code of the local method `printHelper` is included into the method `printList` after in-line expansion. This transformation also works recursively in case local methods again call other local methods.

5.4.5 Establishing Parameter Dependencies

As described in Chapter 4.3, RDSEFFs may contain parameter dependencies for guard expressions, loop iterations numbers, resource demands, and parameter characterisations passed to `ExternalCallActions`. The stochastic expressions (Chapter 3.3.6) specifying these dependencies may only include references to input parameters, but not to local variables. Local variables should not be visible in an RDSEFF, which should not reveal additional information from the implementation of the component and only refer to information specified in interfaces.

However, when Java2PCM finds boolean branch or loop conditions in Java

source code, these may contain references to local variables. Consider the `for`-statement in line 7 of Fig. 5.3(a). It iterates over a local collection called `listCopy`. Thus, the number of elements in this collection specify the number of iterations for the loop. Java2PCM needs to include them into the stochastic expression specifying the number of loop iterations in the RDSEFF. But Java2PCM cannot use the name `listCopy` as it is internal. It can only refer to the input parameter `paramElements` of the public method `printList`, which is converted into a provided service.

Therefore, Java2PCM traces the assignments involving input parameters throughout the code and can then substitute references to local variables in found Java expressions with expressions involving only input parameters references. In the former example, the mapping is straight-forward, as the number of loop iterations can be traced back to the input parameter `paramElements` and Java2PCM can use `paramElements.NUMBER_OF_ELEMENTS` for the stochastic expression defining the parameter dependency on the number of loop iterations.

Algorithm 1: Expand service parameters in expression *exp*

Input: A Java expression *exp* as string
Output: The expression with variables replaced to show affecting parameters

$foundParameters = \{\}$
$intermediateVariables = \{\}$
foreach *simple variable name var in exp* **do**
  if *var is a service parameter* **then**
    $foundParameters = foundParameters \cup \{var\}$
  else
    $assignedExp$ = get expression assigned to var
    $oldFoundParameters = foundParameters$
    Call procedure recursively with argument $exp \leftarrow assignedExp$
    `/*` $foundParameters$ `changed` $\Rightarrow$ var `is connected to parameter */`
    if $oldFoundParameters \neq foundParameters$ **then**
      $intermediateVariables = intermediateVariables \cup \{var\}$
end
foreach *variable interVar in intermediateVariables* **do**
  exp = replace $interVar$ in exp by the expression assigned to $interVar$
end

Figure 5.4: Substituting Local Variables in Expressions

Fig. 5.4 depicts the algorithm that Java2PCM uses when finding an expression in Java code, which is relevant for the RDSEFF and includes local variable references. Before executing the algorithm, Java2PCM has recorded all variable assignments up to this point of the code. With the algorithm, Java2PCM substitutes all local variable names with there current assignments. Because these assignments may again contain local variable names, Java2PCM executes the algorithm recursively

until the assignments refer to service parameters.

Java2PCM executes the algorithm on the former example as follows: the input expression exp is `listCopy`. As `listCopy` is a simple (i.e., local) variable name, Java2PCM executes the algorithm's first loop. `listCopy` is not a service (i.e., input) parameter, thus the Java2PCM enters the else branch. The formerly recorded expression currently assigned to `listCopy` is `elements` from Fig. 5.3(a) line 6. Thus $assignedExp$ is `elements`. Now, Java2PCM executes the algorithm recursively using `elements` as input expression exp. As this is again no input parameter, $assignedExp$ now takes the currently recorded assignment for `elements` as its value, which is `paramElements` (see Fig. 5.3(a) line 2). Because this is an input parameter, it is added to $foundParameters$. When the recursive calls then return and continue their execution after their invocation, the set $foundParameters$ has changed, and `listCopy` and `elements` are added to $intermediateVariables$. Afterwards they are replaced by their assigned expressions so that exp becomes `paramElements`.

Because the algorithm substitutes the left hand side of a variable assignment statement with the right hand side, it produces correct stochastic expressions only in simple cases. However, these case may include arithmetic operations (+,-,*,/) and boolean operations (&&, ||). In cases, where the expression contains for example method invocations, the produced stochastic expression will not be syntactically correct. If input parameters are involved in complex calculations, a whole program slice can be included in the resulting expression. Nevertheless, the included information is useful for the component developer, who has to provide syntactically correct expressions by hand. Therefore, Java2PCM adds the produced expression as a comment to the PCM stochastic expression, which shall specify the parameter dependency, to assist the component developer in manual specification.

5.5 Java2PCM Implementation

Java2PCM's implementation follows three design goals: (i) separation of Java code-related parts and PCM model-related parts to allow their independent evolution, (ii) configurability of the analysis process to let users define the resulting RDSEFF's abstraction level, and (iii) Eclipse integration to reuse existing assets and allow seamless integration with other PCM tools.

Fig. 5.5 shows the design of Java2PCM, which realises the first design goal, the *separation* between Java parts and PCM parts. It is a pipe-and-filter architecture

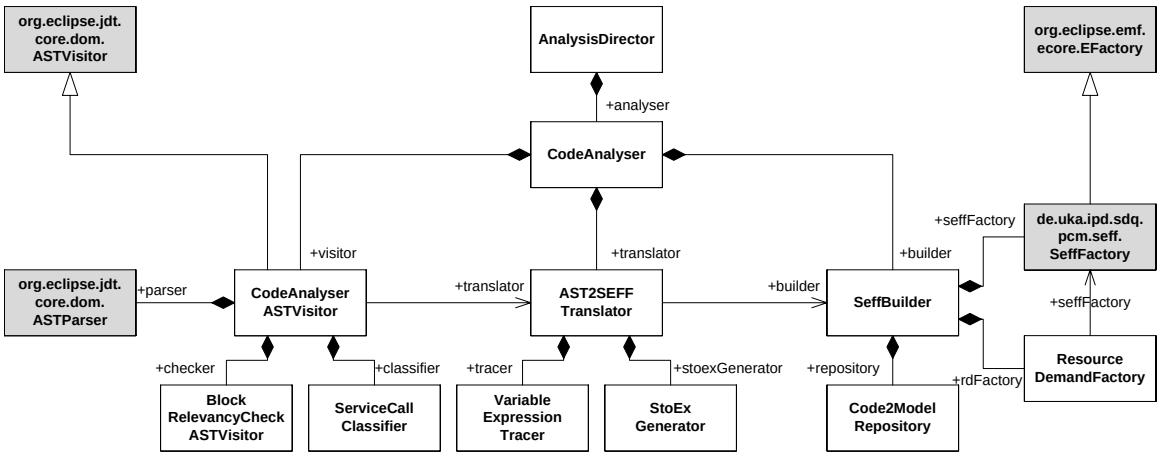


Figure 5.5: Java2PCM: Design Excerpt

with the `CodeAnalysisASTVisitor` producing input for the `AST2SeffTranslator`, which again produces input for the `SeffBuilder`. The `CodeAnalyserASTVisitor` handles Java code and is decoupled from the `SeffBuilder`, which produces PCM model instances.

The `CodeAnalyserASTVisitor` inherits from Eclipse JDT's `ASTVisitor` and is an implementation of the visitor design pattern [GHJV95, pp.331]. After parsing the Java code selected by the user in Eclipse, it traverses its abstract syntax tree (AST), and notifies the `AST2SeffTranslator` upon visiting leafs in the tree. It invokes a second visitor `BlockRelevancyCheckASTVisitor` for each Java code block, which searches for external service calls and resource accesses according to the classification strategy configured by the user and stored in `ServiceCallClassifier`. If such a pre-analysed Java code block does not contain such calls or resource accesses, Java2PCM excludes it from the analysis and RDSEFF build process. If new versions of Java appear, only the `CodeAnalysisASTVisitor` needs to be maintained or replaced.

The `AST2SeffTranslator` performs the mapping from Java elements to PCM elements. It uses the `VariableExpressionTracer` to establish parameter dependencies as described in Chapter 5.4.5. Furthermore, it uses the `StoexGenerator` to translate some Java expressions to syntactically correct PCM stochastic expressions. For example, it translates a Java expression `collection.size()` to the PCM stochastic expression `collection.NUMBER_OF_ELEMENTS`. The translator notifies the `SeffBuilder`, which produces PCM model instances.

The `SeffBuilder` [GHJV95, pp.97] encapsulates invocations of PCM factories [GHJV95, pp.87] and therefore decouples other classes of Java2PCM from the PCM.

It uses the class `Code2ModelRepository` to produce an initial PCM `Repository` with primitive data types, interfaces, components, and empty RDSEFFs. Afterwards, it invokes the `SeffFactory` from the PCM's EMF implementation to create PCM elements, such as `BranchActions`, `ParametericResourceDemands` etc.

The second design goal, the *configurability* of the tool, is realised via a wizard for the classification of method invocations into internal and external calls and a preferences dialog for configuration of global options. Chapter 5.4.1 describes the wizard's functionality. The preferences dialog is integrated into the Eclipse preferences as a separate page. It allows enabling (i) the automatic classification of external service calls, (ii) the automatic creation of `InternalActions` between `ExternalCallActions`, and (iii) classifying only methods implementing services from interfaces as provided services. Furthermore, it offers changing the output path and enabling verbose logging for debugging purposes.

Java2PCM is an Eclipse plug-in, thus realising the third design goal, *Eclipse integration*. It reuses existing Eclipse functionality, such as selection of code, wizards, preferences dialogs, Java code parsing via JDT, Java AST traversal via extended JDT functionality, progress bars, logging to the console. It shall later be integrated into an integrated environment for the reverse engineering of Java code to PCM instances.

5.6 Java2PCM Case Study: CoCoME

During development Java2PCM was tested with several artificial examples ensuring the proper analysis of Java code and proper generation of syntactically correct PCM instances as output. For the final system test, a case study with a realistic component-based system implemented in Java was conducted. As a manually produced PCM instance of this system was available, it was possible to compare it's RDSEFFs to RDSEFFs generated with Java2PCM. This enabled assessing Java2PCM's claimed benefits, i.e., less time consumption and less erroneous model instances than manual modelling, while not altering prediction accuracy.

The system under study, which is called CoCoME (Common Component Modelling Example, [RRMP08]), manages a supermarket chain by providing billing and storage information and allowing the exchange of goods between different stores. Its specification contains more than 20 software components and 8 use cases. The specification includes a set of UML component and sequence diagrams, textual descriptions, and performance annotations according to the UML SPT profile [Obj05b]. CoCoME's Java implementation consists of 5200 lines of code in 97 classes and 40

packages. Although implementing a component-based design, the code is not based on a particular component-based programming model such as EJB or the Spring framework and uses plain Java code instead. Therefore, it is suited for evaluating Java2PCM, which also supports plain Java code.

The case study focusses on use case 8 of CoCoME. This use case allows stores running out of specific goods to query other stores for the availability of these goods. If such a query is successful (i.e., another store has an amount of the desired goods above a certain threshold), CoCoME initiates a shipping procedure delivering the goods to the querying store. The use case only involves interacting services and does not exhibit user participation. It includes network communication between different store servers and the main enterprise server and involves computing a list of stores nearest to the querying store.

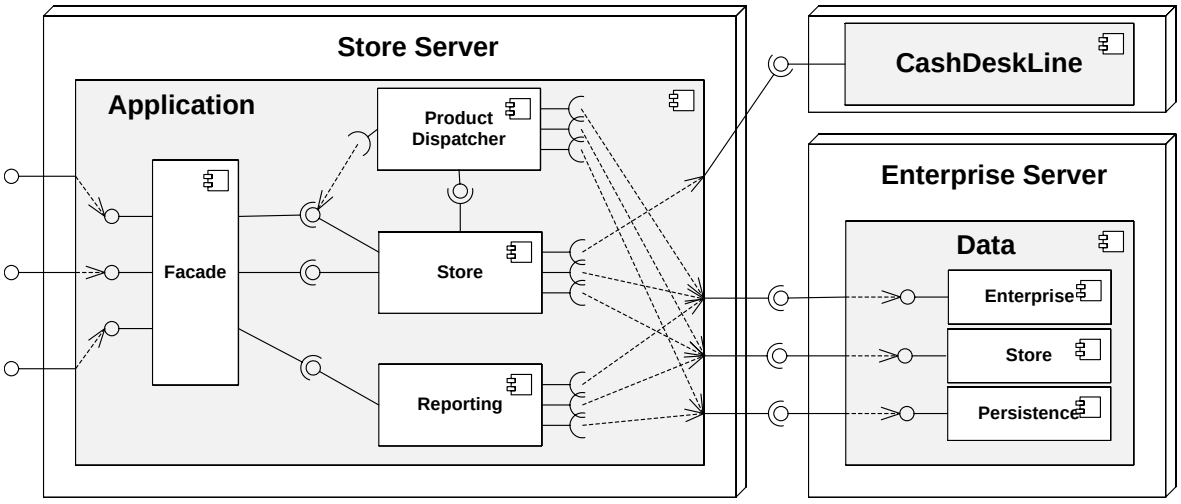


Figure 5.6: Extract from the CoCoME Software Architecture

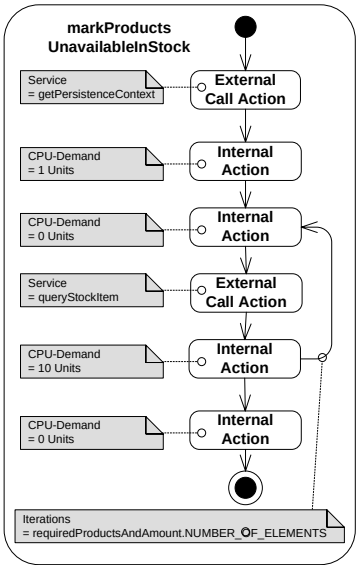
Fig. 5.6 depicts parts of CoCoME architecture participating in this use case. Multiple store servers interact with a single enterprise server, which manages the overall storage information of the supermarket. The components `Application`, `CashDeskLine`, and `Data` are composite components, consisting of nested inner components. The CoCoME specification provides a detailed description of the use case with plain text, a sequence diagram, and performance annotations [RRMP08, pp.12].

Java2PCM generated 10 RDSEFFs from the Java implementation of CoCoME for this use case after manual classification of method calls as internal and external. The produced RDSEFFs included CPU and I/O resource demands set to zero and ini-

tial parameter dependencies. Fig. 5.7 illustrates one of the generated RDSEFFs and its corresponding Java code. The RDSEFFs contains two `ExternalCallActions`, which Java2PCM derived from the code lines 4 and 15. Furthermore, it has a loop iterating over a collection (from code line 13). Java2PCM traced the iterator back to the input parameter `requiredProductsAndAmount` and set a parameter dependency for the number of loop iterations (`requiredProductsAndAmount.NUMBER_OF_ELEMENTS`).

```
1  public void markProductsUnavailableInStock(
2          ProductMovementTO requiredProductsAndAmount)
3          throws RemoteException, ProductNotAvailableException {
4      PersistenceContext pctx = persistmanager.getPersistenceContext();
5      TransactionContext tx = null;
6      try {
7          tx = pctx.getTransactionContext();
8          tx.beginTransaction();
9
10         Iterator<ProductAmountTO> productAmountIterator =
11                 requiredProductsAndAmount.getProducts().iterator();
12         ProductAmountTO currentProductAmountForDelivery;
13         while(productAmountIterator.hasNext()) {
14             currentProductAmountForDelivery = productAmountIterator.next();
15             StockItem si = storequery.queryStockItem(
16                     requiredProductsAndAmount.getDeliveringStore().getId(),
17                     currentProductAmountForDelivery.getProduct().getBarcode(),
18                     pctx);
19             if(si == null) {
20                 throw new RuntimeException( <...> );
21             }
22             // set new remaining stock amount:
23             si.setAmount( si.getAmount() -
24                     currentProductAmountForDelivery.getAmount() );
25             System.out.println( <...> );
26         }
27         tx.commit();
28     } catch (RuntimeException e) {
29         <...>
30     }
31 }
```

(a) Java Code



(b) RDSEFF

Figure 5.7: CoCoME Service `markProductsUnavailableInStock`

The `InternalActions` result from code not containing external service calls. Java2PCM generate CPU resource demands with an amount of zero for these actions. For this case study, they were completed manually with information from the performance annotation specification of this use case [RRMP08, pp.19]. In the future this step shall be automated by determining the resource demands automatically via dynamic program analysis.

To enable running the simulation [BKR07], a manually specified `ResourceEnvironment`, `System`, `Allocation`, and `UsageModel` were added to the RDSEFFs to complete a full PCM instance. The whole specification process including running Java2PCM and debugging the models took 4 hours compared to the 40 hours formerly needed for the fully manually specified model of this use case. Running the simulation yielded a large amount of measures for this use case's end-to-end response time. Fig. 5.8 visualises them as a cdf (dark line). The predicted

measures' median was 6200 ms. The same figure also shows the predicted measures for the manually built model from [KR08] (bright line). Their median is at 6150 ms. Both curves are widely overlapping, therefore the prediction results are very similar. This shows that it was possible to achieve roughly the same prediction results with generated models and the manually specified models. Notice that both cdfs are only predictions and there is no comparison to actual measurements with an implementation, because the purpose of this case study was not to assess the prediction accuracy achievable with the available information, but only to compare predictions with manual and semi-automatic specified models given the same type of input information.

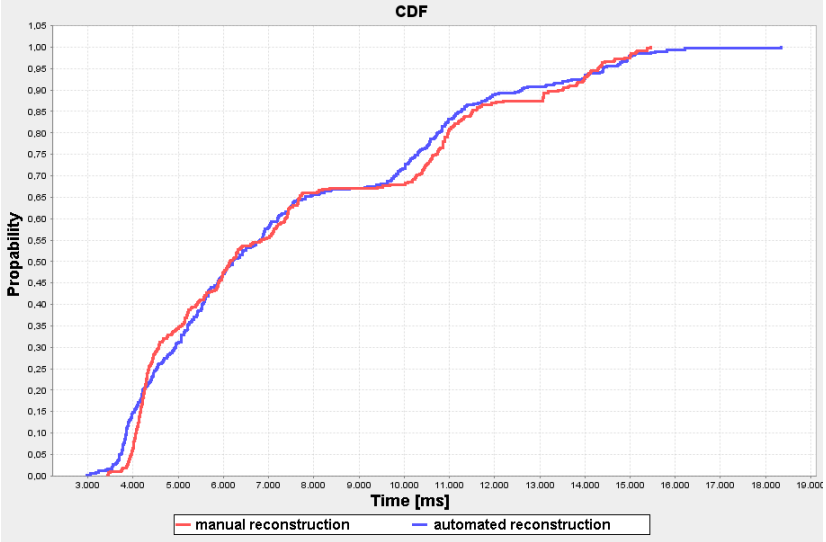


Figure 5.8: Comparison of manual and automated reconstruction

Besides the quantitative prediction results, having a manual and a generated specification enabled comparing them *qualitatively*. The generated models were structurally more complete than the manually specified ones, because several abstractions had been made during manual specification by hand for example by merging some `InternalActions` or omitting unimportant `ExternalCallActions`. While the generated models did contain more `InternalActions`, this did not change the prediction results, because their resource demands were set to zero and only the manually added amounts influenced the predictions.

Java2PCM could trace *parameter dependencies* correctly only in very few cases, because they often included complex calculations with method calls, which were not directly convertible to PCM stochastic expressions. However, the program slices

generated by Java2PCM's parameter tracing, which were included as comments into the corresponding stochastic expressions, gave hints for the manual abstraction of the dependencies. This considerably sped up the modelling procedure.

Still, the Java2PCM's implementation has several *open issues*: It does not trace output parameter characterisations so far. There is no support for composite components yet, which shall later be provided by the component detection tool run before Java2PCM. Java2PCM does not support tracing parameter dependencies correctly, if they contain nested method calls (e.g., `foo().bar().foobar()`). Furthermore, Java2PCM only works with primitive and collection data types and does not deal correctly with composite data types. Finally, Java2PCM's integration into the tool chain envisioned for the complete reverse engineering approach (Chapter 5.3) is missing.

There are also some *general limitations*, which restrict static code analysis and require the combination with dynamic program analysis. The halting problem prevents from determining loop iteration numbers or dependencies to loop iteration numbers in the general case. Only simple cases of determining loop iteration numbers are supported by Java2PCM. Polymorphism involves data types only known during run time, therefore limiting static analysis. Using native code instead of the Java API might lead to resource demands or external call actions not determinable by a static analysis plainly for Java code.

5.7 Summary

This chapter tackled the problem of automatic performance model generation from implemented software components. First, it briefly survey different methods suitable for model generation from code including static code analysis, program slicing, dynamic analysis, symbolic execution, and prototyping. Afterwards, this chapter proposed hybrid approach consisting of static and dynamic analysis to create RDSEFFs from arbitrary Java code. The static code analysis part of this process was implemented for this thesis in the tool Java2PCM. This tool creates initial RDSEFFs from Java code, while applying the formerly described abstractions of RDSEFFs. Java2PCM is able to correctly reconstruct control flow structures in many cases, but cannot determine resource demands, which must be added later via dynamic analysis. This approach was evaluated by creating RDSEFFs for the COCOME architecture, which were suitable for performance predictions after some manual additions. While Java2PCM significantly reduced the effort for creating the models, the subse-

quent performance prediction with these models achieved an accuracy comparable to the manually specified models.

Chapter 6

Model-Transformation from Software Domain to Performance Domain

6.1 Model-Transformation Process

To conduct a performance prediction with a PCM instance and derive performance metrics such as response time, throughput, and resource utilisation, a transformation into a performance model (such as a queueing network, stochastic process algebra, or stochastic Petri net) is necessary. This work does not introduce a new performance model, but implements transformations into existing performance models to reuse existing analytical solvers and simulation methods. This chapter will describe these transformations and the corresponding model solvers.

Before transforming a PCM instance into a performance model, tools have to resolve the included parametric dependencies, which have been added to the model in this work. Parametric dependencies enable different developer roles to specify their models independently. However, most known performance models do not support parametric dependencies as they focus on the timing aspects of a system. The QPNs used in Chapter 4.4 to specify the semantics of the PCM are an exception. However, as explained before their analysability is limited due to their high expressiveness.

In order to reuse efficient solvers for existing performance models, this work first solves parametric dependencies present in a PCM instance by combining the information from component developers, software architects, system deployers, and domain experts. Fig. 6.1 depicts the process of this model transformation. The first step, i.e., the solution of parametric dependencies, does not depend on the targeted

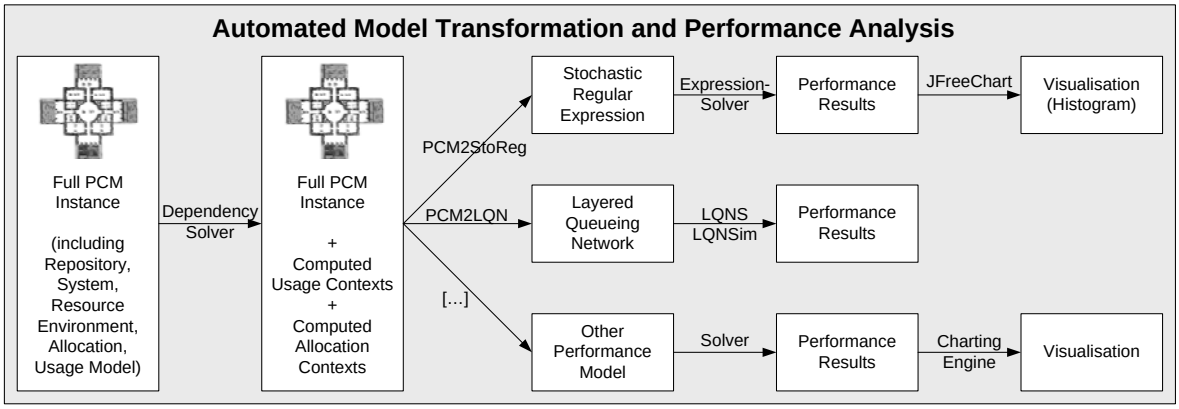


Figure 6.1: Model Transformation Process Model

performance model, and is therefore the same regardless of the performance model transformation. Chapter 6.2 describes in detail the tool `DependencySolver` that implements this step. As depicted in Fig. 6.1, it uses so-called "computed contexts" to store the information that results from solving parametric dependencies.

After solving parametric dependencies, different performance model transformations are possible. The developer role responsible for conducting the performance prediction must choose a suitable performance model. The selection can depend on the desired performance metrics. For example, if a performance analyst is interested in distribution functions for response time, a performance model supporting such functions must be used. The selection can also depend on the time available for the prediction. For example, if management has granted the performance analysts a larger time frame for the prediction, a performance model with a precise, but long-running simulation solver can be selected instead of a possibly less accurate, but fast analytical solver. Jain [Jai91] discusses the general benefits and drawbacks of different performance models and solvers. Becker et al. [BKR08] compare different solvers for PCM instances.

In the context of this work, transformations into two performance models have been implemented. The first is a transformation into stochastic regular expressions, which support response time predictions with arbitrary distribution function, but are only applicable in single-user scenarios (Chapter 6.3). The second is a transformation into LQNs, which support multi-user scenarios, but are limited to exponential distribution functions and mean-value analysis (Chapter 6.4). Happe [Hap08] defines a transformation from PCM instances with solved dependencies into a stochastic Process algebra.

The model transformations in context of this work have been implemented using Java and the Eclipse Modelling Framework [Eclb]. No specific model transformation engine (e.g., based on QVT, ATL) was used due to immature tool support. Because of the involved mathematical calculations, a transformation using QVT would have to use black box implementations (in Java) outside the model transformation language for efficiency reasons. Furthermore, the transformations do not rely on an intermediate performance modelling language (such as CSM [PW06] or KLAPER [GMS07b]), because these languages include too hard assumptions and so far have limited tool support.

6.2 Dependency Solver

The `DependencySolver` (DS) is a tool to substitute parameter names inside PCM stochastic expressions with characterisations originating from the usage model. In the usage model, the domain expert has to specify a variable characterisation (e.g., a constant or probability distribution) for each `RequiredCharacterisation` specified by component developers in `Interfaces` (see Chapter 4.1.4).

The DS propagates these characterisations through all elements of a PCM instance and inserts them into guard specifications, parametric loop iterations, parametric resource demands, and parameter usages specified by the component developer. Then, it solves the resulting stochastic expressions, so that they become constant values or probability distributions, and stores them, so that they can be used for a transformation into a performance model.

This subsection first describes the expected input and produced output of the DS. Then, it describes the traversal of PCM instances, which depends on the evaluated parameter characterisations. Finally, it shows the process of solving dependencies, before giving an example.

6.2.1 Input and Output

As input, the DS expects a valid PCM instance, i.e., all developer roles must have contributed their part of the model. As output, the DS produces a set of so-called `ComputedUsageContexts` and `ComputedAllocationContexts`. These are decorator models [Ecla] for the PCM instance and store stochastic expressions resulting from solving the parametric dependencies. Transformations from PCM instances to performance models use these stochastic expressions to create per-

formance model annotations, for example for branch probabilities or resource demands.

The DS creates a `ComputedUsageContext` (Fig. 6.2) for each usage of an RDSEFF in an `AssemblyContext`. This means that the DS produces two different `ComputedUsageContexts` if the same RDSEFF in the same `AssemblyContext` is invoked twice with different parameter characterisations. A `ComputedUsageContext` stores `BranchProbabilities`, `LoopIterations`, and a set of parameter characterisations.

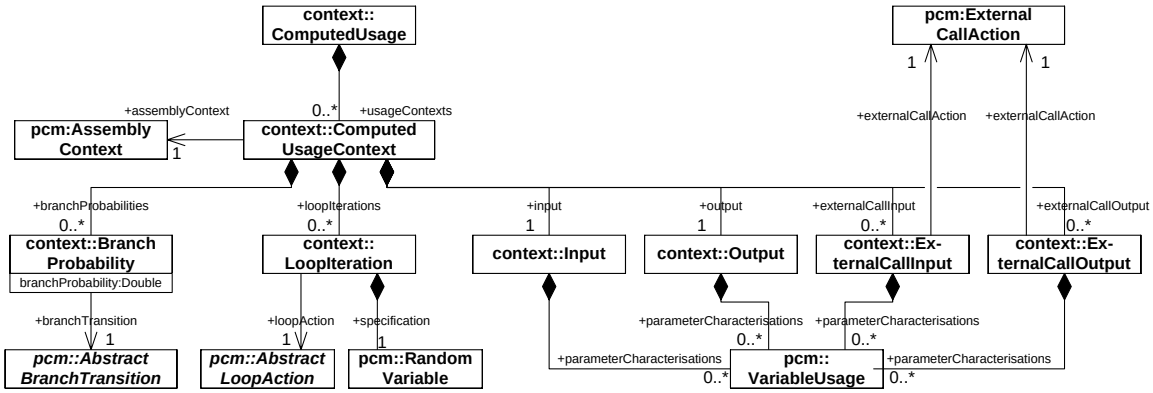


Figure 6.2: Computed Usage Context

A `BranchProbability` results from a solving parametric dependency in a `GuardedBranchTransition` or simply from copying the branch probability from a `ProbabilisticBranchTransition`. A `LoopIteration` holds a pmf for the number of loop iterations and results from solving parametric dependencies to stochastic expressions specifying loop iterations (for `LoopActions`) or by using the number of elements characterisation of an input parameter (for `CollectionIteratorActions`).

The `ComputedUsageContext` distinguishes between different kinds of parameter characterisations (Fig. 6.2):

- `Input` holds solved characterisations of IN and INOUT parameters of the current RDSEFF's service.

- `Output` holds solved characterisations of OUT and INOUT parameters of the current service, if the component developer has set them in an RDSEFF with `SetVariableActions`.

- `ExternalCallInput` stores solved characterisations for IN and INOUT parameters of a specific `ExternalCallAction` in the current RDSEFF.

- `ExternalCallOutput` stores solved characterisations for OUT and INOUT parameter of a specific `ExternalCallAction` in the current RDSEFF.

These parameter characterisations are not useful in a performance model, but transformations to performance models use them to retrieve the correct `ComputedUsageContext` from all computed contexts when traversing the PCM instance, because an `Input` and the current `AssemblyContext` determine it unambiguously as explained above. When comparing two `Inputs`, their included `VariableUsages` are compared to each other.

The DS stores resource demands with solved dependencies in `ComputedAllocationContexts`. This is a separate model from the `ComputedUsageContexts`, because the source of information is the component developer, software architect, domain expert, and the system deployer, whereas the source of information for `ComputedUsageContexts` are only the component developer, software architect, and the domain expert. Furthermore, the model for solved resource demands may change in the future, if a more refined resource model is introduced.

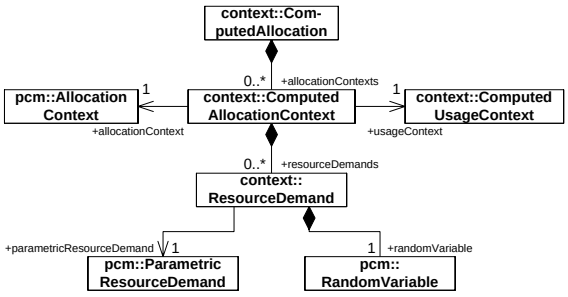


Figure 6.3: Computed Usage Context

The `ResourceDemands` in `ComputedAllocationContexts` result from substituting variable references in a `ParametericResourceDemand` with parameter characterisations, dividing the resulting specification by the processing rate of the referenced `ProcessingResourceSpecification` and then solving the whole stochastic expression. The result is a pdf specifying the processing time demanded from a particular resource by a particular `InternalAction`.

6.2.2 Model Traversal

The DS implements two visitors, which traverse the PCM instance given as input and produce the `ComputedUsageContexts` and `ComputedAllocationContexts` when arriving at the final actions of RDSEFFs. Fig. 6.4 sketches the traversal process involving a usage scenario and several RDSEFFs. The thick arrows indicate the invocation of another visitor for an RDSEFF upon arriving at an `EntryLevelSystemCall` or `ExternalCallAction`.

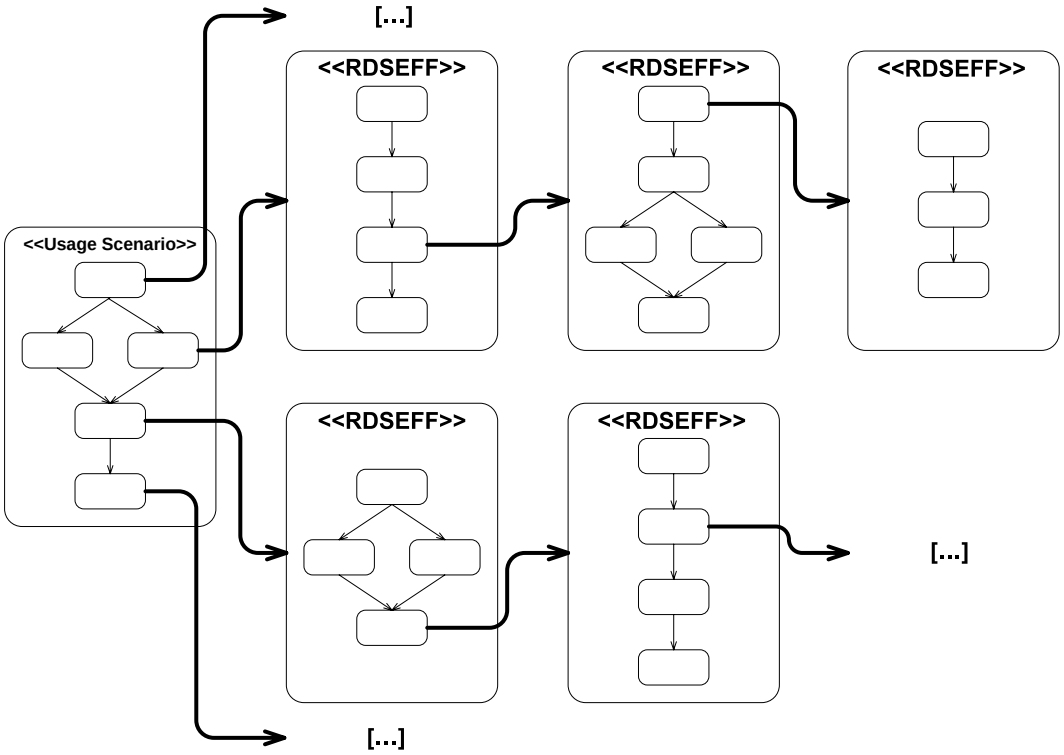


Figure 6.4: DSolver: Traversing a PCM Instance

The first visitor traverses all usage scenario instances of the usage model included in the PCM instance. Upon arriving at an `EntryLevelSystemCall`, it solves dependencies on its input parameter characterisations, creates a new `ComputedUsageContext`, and adds the solved characterisations to its `Input`.

If the domain expert has specified an `EntryLevelSystemCall` inside a `Loop`, the DS creates only a single `ComputedUsageContext` for this call. The PCM forbids parameter characterisation to be bound to the iteration number of a loop, therefore the input parameter characterisations for such calls inside a loop do not change. Thus, the DS only needs to create a single `ComputedUsageContext` in this case.

The usage scenario visitor uses the provided role referenced by the EntryLevelSystemCall to determine the called AssemblyContext and the corresponding RDSEFF. It then creates a second visitor for RDSEFFs and initiate it's execution on the first action of this RDSEFF. If this visitor arrives at an ExternalCallAction, it finds a ComputedUsageContext for the RDSEFF corresponding to this external call or creates a new one. Just as the usage model visitor, it solves the parameter dependencies on input parameter characterisations and adds them to the ComputedUsageContext's Input. Afterwards, it continues traversing the next RDSEFF.

If the RDSEFF visitor arrives at a SetVariableAction it solves dependencies on the attached variable characterisations and adds them to the current ComputedUsageContext's Output. If the return value or output parameter had been characterised by a SetVariableAction before (for example in a sequence or loop), the visitor overwrites the old characterisation. SetVariableActions inside ForkedBehaviours are not supported so far due to the immaturity of the ForkAction. If SetVariableActions occurs in different branched behaviours, the visitor uses the formerly computed branching probability to derive a pmf characterising the return value or output parameter before adding it to the Output.

The RDSEFF visitor traverses all ResourceDemandingBehaviours of AbstractBranchTransitions even if their probability is zero. As in the usage model, the RDSEFF visitor traverses ResourceDemandingBehaviours inside loops only once. Finally, if the RDSEFF visitor arrives at the final action of an RDSEFF, it saves the current ComputedUsageContext and continues the traversal at the ExternalCallAction in the calling RDSEFF, which originally invoked the RDSEFF.

It switches back to the ComputedUsageContext of that RDSEFF and uses the Output from the former ComputedUsageContext to solve parametric dependencies on output parameter characterisations of the ExternalCallAction. The visitor then adds these solved characterisations to the ExternalCallOutput of the current ComputedUsageContext. When traversing subsequent actions, these characterisations in addition to the Input characterisations may be used to solve parametric dependencies in stochastic expressions as described in the following.

6.2.3 Solving Dependencies

In addition to solving dependencies within input and output parameter characterisations, the RDSEFF visitor of the DS resolves parametric dependencies within branching guards, parametric loop iteration numbers, and parametric resource demands. It substitutes parameter references used by component developers with variable characterisations provided by domain experts or other component developers. Fig. 6.5-6.6 depict an example of this process, which in this case includes 6 steps.

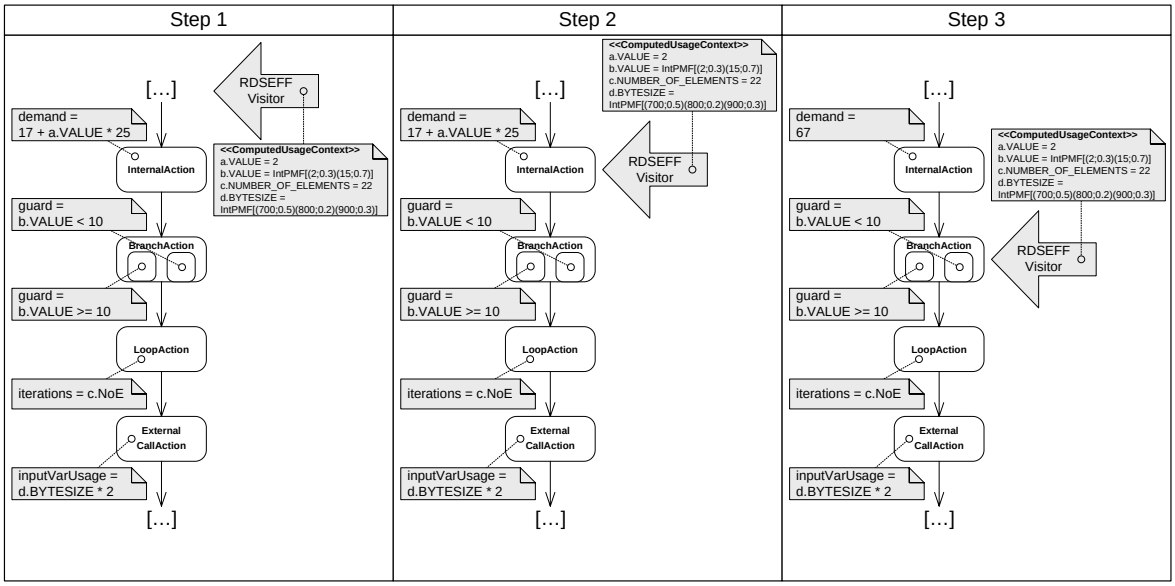


Figure 6.5: DSolver: Solving Parametric Dependencies (1/2)

In step 1, all model annotations contain parametric dependencies, and the RDSEFF visitor (indicated by the large arrow) is located at the predecessor action of the `InternalAction`. In step 2, the RDSEFF visitor has moved downwards to the `InternalAction` and now solves the parametric dependency in the attached `ParametricResourceDemand`. It substitutes the variable reference `a.VALUE` with the (solved) parameter characterisation in the current `ComputedUsageContext's Input` or `ExternalCallOutput`.

In the example the `ComputedUsageContext` contains the characterisation `2` for `a.VALUE`. Therefore, substituting the variable reference yields a stochastic expression `17+2*25`. Using the operations defined in Chapter 3.3.6 such an expression can be solved to a constant or a probability function if it involves probability distribu-

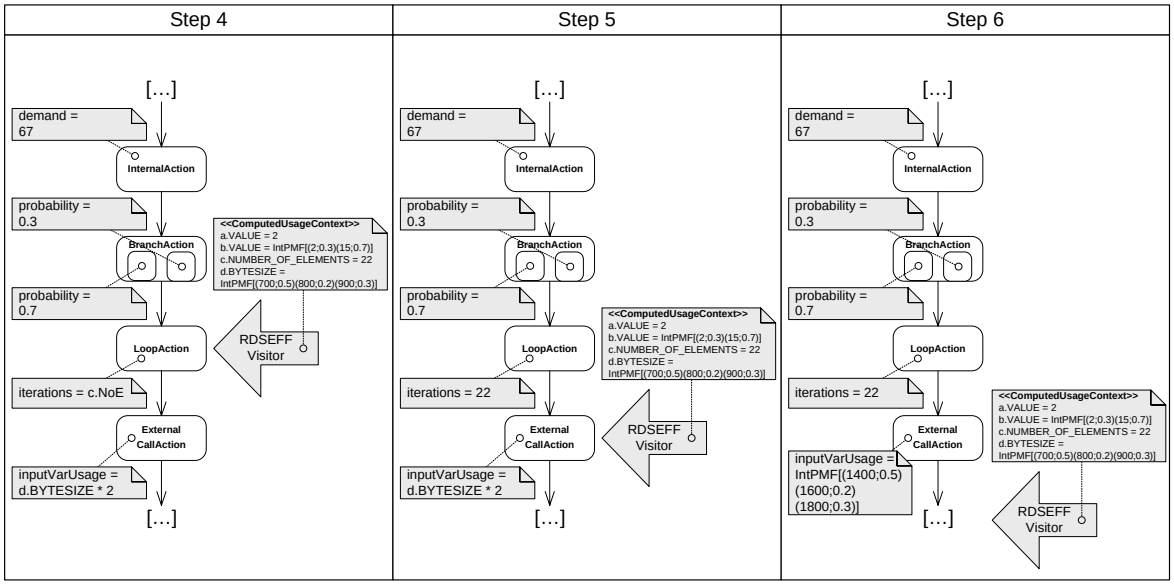


Figure 6.6: DSolver: Solving Parametric Dependencies (2/2)

tions. In this example, it results simply in the constant `67`. As explained before, the DS also includes the referenced resource's processing rate and the expression above by it. In this example the processing rate is simply `1.0`, therefore the DS resolves the `ResourceDemand` of the `ComputedAllocationContext` for this `InternalAction` to the constant `67`, which is depicted in step 3. This resource demands can later be used in a performance model, for example as the service demand at a service center in a queueing network.

In step 3, the RDSEFF visitor processes the guard specifications of the `GuardedBranchTransitions` of the `BranchAction`. The `ComputedUsageContext` contains a pmf for `b.VALUE`. The RDSEFF visitor can map the contained probabilities (0.3, 0.7) directly to the branch probabilities, as visible in step 4. Branch probabilities can later be used in a performance model, for example as the firing weight of a stochastic Petri net.

In step 4 and step 5, parametric dependencies on loop iterations numbers and parameter usages are solved in the same manner as explained above.

Fig. 6.1 illustrates the algorithm for solving parametric dependencies. It gets the `ComputedUsageContext` and a `Variable` as input, and returns a solved stochastic expression as output. First, it retrieves all input parameter characterisations and output parameter characterisations from former external calls of the RDSEFF and puts them into a list (line 7-9). Then, it searches the list for the `Variable-`

Usage given as input by using its full qualified name (line 11-22). If found (line 13), this VariableUsage may contain multiple characterisation (e.g., BYTESIZE and NUMBER_OF_ELEMENTS). Therefore, the algorithm checks whether the needed characterisation is specified (line 16). If this is the case, the algorithm retrieves its stochastic expression as given in the ComputedUsageContext and returns it.

```
1  Input:
2  ComputedUsageContext ctx // current computed usage context
3  Variable variableToSolve // variable found in a stochastic expression
4  Output:
5  Expression               // resolved stochastic expression
6
7  List varList = new ArrayList();
8  varList.addAll(computedUsageContext.getInput()); // add inputs
9  varList.addAll(computedUsageContext.getExternalCallOutput()); // add former outputs
10 String wantedVariableName = getFullName(variableToSolve);
11 for (VariableUsage ctxVar : varList){
12   String currentVariableName = getFullName(ctxVar);
13   if (currentVariableName.equals(wantedVariableName)){
14     List varCharList = ctxVar.getVariableCharacterisation();
15     for (VariableCharacterisation ctxVarChar : varCharlist) {
16       if (ctxVarChar.getType() == variableToSolve.getType()){
17         return ctxVarChar.getExpression();
18       }
19     }
20     // error message: variable characterisation missing in usage context
21   }
22 }
23 // error message: variable missing in usage context
```

Listing 6.1: Solving Parametric Dependencies

Afterwards the RDSEFF visitor needs to solve the stochastic expression resulting from substituting the variable reference. The semantics of this solution are given in Chapter 3.3.6.

6.2.4 Context Wrapper

For convenient implementation of model transformations in Java from PCM instances to performance models, the DS provides a so-called ContextWrapper. It hides all specified and computed context models from the transformation and assists the traversal of a PCM instance. A transformation can instantiate a new ContextWrapper upon visiting an EntryLevelSystemCall or External-CallAction as it is specific for each RDSEFF call.

Listing 6.2 illustrates a part of the services provided by the ContextWrapper. Transformations must instantiate a ContextWrapper initially when visiting the

first `EntryLevelSystemCall` by calling its constructor and passing a reference to the current PCM instance, which already includes the specified contexts as well as the computed contexts from a former run of the DS. Thus, from an `EntryLevelSystemCall` and the given PCM instance, the `ContextWrapper` can retrieve the called assembly context, allocation context, computed usage context, and computed allocation context internally.

The `ContextWrapper` also includes functions to retrieve the RDSEFF called by an `EntryLevelSystemCall` or `ExternalCallAction`, which a transformation needs to continue traversing a PCM instance. These functions (`getNextSEFF`) hide the context-dependent traversal through the model via delegation and assembly connectors from the transformation.

```
1  // Functions assisting the traversal of a PCM instance
2  public ContextWrapper(EntryLevelSystemCall elsa, PCMInstance pcm);
3  public ContextWrapper getContextWrapperFor(EntryLevelSystemCall elsa);
4  public ContextWrapper getContextWrapperFor(ExternalCallAction eca);
5  public ServiceEffectSpecification getNextSEFF(EntryLevelSystemCall elsc);
6  public ServiceEffectSpecification getNextSEFF(ExternalCallAction eca);
7
8  // Functions producing annotations for performance models
9  public double getBranchProbability(AbstractBranchTransition abt);
10 public ManagedPMF getLoopIterations(AbstractLoopAction ala);
11 public ManagedPDF getTimeConsumption(ParametricResourceDemand prd);
12 public ManagedPDF getDelayOnLinkingResource(ExternalCallAction eca,
13                         CommunicationLinkResourceSpecification clrs);
```

Listing 6.2: API of the Context Wrapper

When a model transformation visits RDSEFF actions, it may call the `ContextWrapper` for performance annotations, such as branch probabilities, loop iteration numbers, or timing values. This information is not contained in the parametrised RDSEFF, but only in the computed contexts. The `ContextWrapper` retrieves the information from the computed contexts given for example an `AbstractBranchTransition` or `ParametricResourceDemand`.

6.2.5 Computational Complexity

The following estimates the time and memory requirements for running the DS on a PCM instance, which depends on the number of contexts it has to compute. Therefore, the following first estimates the maximum number of `ComputedUsageContexts` and `ComputedAllocationContexts` for a given PCM instance.

For estimating the number of `ComputedUsageContexts`, let R be the number of RDSEFFs of the PCM instance, C_{assCtx} the maximal number of assembly contexts

per component, and I the maximal number of invocations of a specific RDSEFF with different input parameter characterisations. Then, the maximal number of created `ComputedUsageContexts` for a PCM instance is:

$$C_{compUsgCtx} = R * C_{assCtx} * I$$

Each invocation I with different input characterisations influences the `Input` of a `ComputedUsageContext`, therefore a new `ComputedUsageContext` needs to be created. The number of `ComputedUsageContexts` depends on the number of assembly contexts, because the same RDSEFF with the same input in different assembly contexts may receive different `ExternalCallOutput` from connected components, which then changes the `ComputedUsageContext`.

The number of `ComputedUsageContexts` does *not* depend on the number of loop iterations specified in RDSEFFs, because the PCM does not allow input parameter characterisations of `ExternalServiceCalls` to change within a loop. Therefore, multiple invocations of an `ExternalServiceCall` within a loop inside a RDSEFF do not increase I, which increases only for *different* input parameter characterisations. Furthermore, the number of `ComputedUsageContexts` does not depend on the number of allocation contexts of a component, because the information inside a `ComputedUsageContext`, such as loop iteration numbers and branch probabilities, does not depend on the `ResourceContainer` a component is allocated on.

For estimating the number of `ComputedAllocationContexts`, let C_{allCtx} be the maximal number of allocation contexts per assembly context. Then, the maximal number of `ComputedAllocationContext` for a PCM instance is:

$$C_{compAllCtx} = C_{compUsgCtx} * C_{allCtx}$$

The number of `ComputedAllocationContexts` depends on the `ComputedUsageContext`, because `ParametricResourceDemands` may include parameter characterisations. It depends on the number of allocation contexts, because the processing rate of the referenced resources is used for the calculation of `ResourceDemands` inside `ComputedAllocationContexts`.

However, in the current version of the PCM, C_{allCtx} is always 1, as a concept for replication is missing and it is not possible to allocate an `AssemblyContext` to different resource containers. Therefore, the maximum number of `ComputedAllocationContexts` is the same as the maximum number of `ComputedUsageContexts`.

The calculation of a computed allocation or usage context may lead to a limited number of convolutions when solving stochastic expressions that include operations on probability distribution functions. A straight-forward implementation of a discrete convolution has a computational complexity of $O(N^2)$, where N is the number of sampling points in the involved distribution functions. As shown in [FBH05], Fast Fourier transformation provides an efficient way of computing a convolution, because the computational complexity reduces to $O(N)$ in the frequency domain. However, the Fourier transformation from the timing into the frequency domain has a complexity of $O(Nlog(N))$. Thus, the total cost of a convolution is in $O(Nlog(N))$ using Fast Fourier transformation.

Let X be the maximum number of convolutions involved in creating a computed context for a PCM instance. Then, the total complexity of executing the DS on an arbitrary PCM instance is in:

$$O((C_{compUsgCtx} + C_{compAllCtx})XNlog(N))$$

It can be assumed that the number of computed contexts, as well as the number of convolutions is much smaller than the number of sampling points of the involved probability distribution function:

$$C_{compUsgCtx}, C_{compAllCtx}, X << N$$

This reduces computational complexity to $O(Nlog(N))$.

6.3 Transformation to Stochastic Regular Expressions

6.3.1 Overview

The Stochastic Regular Expression (SRE) model is an analytical performance model in the class of semi-Markov processes [Tri01]. It consists of a discrete time Markov-chain (DTMC) to model state transitions, but the sojourn time in each state can follow arbitrary probability distributions instead of being limited to exponential distributions as in Markov chains. Furthermore, SREs are hierarchically structured and do not allow cycles in the embedded DTMC for more accurate predictions. Chapter 6.3.3 will provide the syntax and semantics of SREs, afterwards Chapter 6.3.4 shows how to compute overall sojourn times with SREs.

Only a partial transformation of PCM instances to SREs is possible, because of the model's limited expressiveness. The transformation is straight-forward, as the

control flow modelling of PCM instances and SREs are closely aligned. Chapter 6.3.5 will describe the transformation PCM2SRE.

While allowing accurate predictions by supporting arbitrary distribution functions for timing values, SRE are limited to analysing single-user scenarios. They do not include queues or control flow forks, and cannot express contention effects due to concurrent requests. However, they provide a fast method of producing performance predictions during early development stages, as they are usually more quickly solved than running a simulation. Chapter 6.3.6 discusses the assumptions underlying SREs in detail. The SRE model will be used for a performance prediction in a case study in Chapter 7.3.3.

6.3.2 Background

The Palladio group has developed SREs for several years. Reussner et al. extended the service effect finite state machines from [Reu01a] with transition probabilities and reliability measures to Markov chains [RPS03] and conducted reliability predictions for component-based software systems. Later, Reussner et al. [RFB04] added arbitrary timing delays to the model to conduct performance predictions, thereby creating semi-Markov chains . Firus et al. [FBH05] increased the efficiency of the solution algorithm by using Fast Fourier Transformations to calculate the involved convolutions. Happe [Hap04] converted the semi-Markov chains into SREs to enable an easy implementation of the solution algorithm. Koziolek et al. [KF06] extended the model to allow an arbitrary distributed number of loop iterations, thus removing the former restriction to a geometrically distributed number of iterations.

In order to remove the restriction to sequences, alternatives, and loops in the control flow, Happe et al. [HKR06] introduced an initial concept to express forks, which also includes the available number of processors into the prediction. However, this extension suffered from hard assumptions and worked only in restricted cases. For example, it does not reflect accesses to memory busses, cache thrashing, or automatic core switching on multi-core CPUs. Currently, Happe [Hap08] is extending the SRE model to a full stochastic process algebra, which supports contention effects and concurrent behaviour. In this thesis, there is no description of SRE concepts concerning concurrent behaviour. The focus here is on control flow including only sequences, alternatives, and loops. For the expression of concurrent behaviour, the interested reader is referred to [Hap08].

6.3.3 Syntax and Semantics

The definition of SREs is based on semi-Markov processes and related to stochastic process algebras. First, the syntax and the semantics are defined formally, before a discussion of the model follows. The syntax of a SRE is given by the following definition:

Definition 18 Stochastic Regular Expression

Let a be a terminal symbol from an alphabet Σ, let P and Q be non-terminal symbols, $\pi \in [0,1]$ be a probability, and $l : \mathbb{R} \to [0,1]$ be a pmf for a number of loop iterations. Then, the syntax of a **Stochastic Regular Expression** (**SRE**) is defined by the following grammar in BNF:

$$P := a \mid P \cdot Q \mid P +_\pi Q \mid P^{*(l)}$$

In addition, each terminal symbol a has an associated random variable X_a characterised by a pdf $f_a(t)$, which defines a sojourn time. For each $l(i)$ pmf defining a number of loop iterations, it must hold that $\exists N \in \mathbb{N}_0 : \forall i > N : l(i) = 0$, which bounds the number of loop iterations.

The semantics of SREs are given as follows:

- Symbol (a): models a sojourn time given by the random variable X_a. It represents the time consumption needed for some operation of a software system.
- Sequence ($P \cdot Q$): models that first P is executed, afterwards Q is executed. The dot can be omitted when writing an SRE. It represents the time consumption of successive operations of a software system.
- Alternative ($P +_\pi Q$): models that either P is executed with probability π or that Q is executed with probability $1 - \pi$. It represents time consumption for some operation of a software system selected based on a probabilistic choice.
- Loop ($P^{*(l)}$): models that P is executed once with probability $l(1)$, or twice with probability $l(2)$, or n-times with probability $l(n)$. It represent the time consumption of an operation of a software system, which is executed repetitively.

The model focuses on the *time consumption* of software systems. It does not distinguish between different resources. Therefore it does not directly enable to derive

system-oriented performance metrics, such as resource utilisation. However, it allows predicting the overall execution time of a service or software system as an arbitrary distribution function by combining the pdfs of individual symbols as described in the next chapter.

Expressing time consumption of distributed information systems with arbitrary distribution functions is desirable, because such systems often exhibit a complex behaviour due to many influencing factors, such as the hardware, operating system, middleware, concurrently running software, different usages, external services connected via the Internet etc. Their time consumption is often not adequately captured with mean values and standard deviations or even common probability distributions, such as an exponential or Erlang distributions (also see [BR06]).

Predicted timing values for a software system as arbitrary distribution functions provide more information to a performance analyst. They may help identifying patterns in the responsiveness of a software system and provide rationale to direct the search for causes of performance problems.

Besides using arbitrary distribution functions, the model expresses the control flow in a software system in a structured manner by using regular expressions. Unlike plain Markov chains, regular expressions do not allow arbitrary cycles in the control flow. They require to make all control flow loops explicit by using the Kleene star operator. This forbids for example loops with multiple entrance points or intertwined behaviour.

It furthermore enables directly specifying the number of loop iterations, whereas Markov chains model loops with control flow cycles (i.e., backward references) and a reentry probability p and an exit probability $1 - p$. This only indirectly expresses the number of loop iterations and binds them to a geometrical distribution, which almost never reflects behaviour of realistic software systems well [DG00, KF06].

For accurate modelling, SREs allow expressing the number of loop iterations with pmfs having arbitrary distribution functions. However, the number of iterations needs to be bounded, so that the modelled time consumption is finite and that passage time metrics can be determined from the model. Other common performance models, such as the execution graphs by Smith et al. [Smi02] or UML models annotated according to the SPT profile [Obj05b], only allow mean values for the number of loop iterations.

6.3.4 Overall Sojourn Time Solution

SREs allow the prediction of an overall sojourn time for the modelled time consumptions, for example to gain the end-to-end response time of a usage scenario as a distribution function. The following describes the individual computations involved in this process.

As defined before, the sojourn time of a **terminal symbol** a is the random variable X_a, which is characterised by a pdf $f_a(t)$. The pdfs' of all terminal symbols in a SRE are assumed independent and identically (iid) distributed, which is necessary for the later computations. To add the execution time of a terminal symbol to the overall execution time consider the following. If X_t is the random variable for the already evaluated terminal symbols when $a \cdot P$ begins, then the overall execution time is $X'_t := X_t + X_a$, when a finishes. Afterwards, $a \cdot P$ behaves like P at time X'_t, meaning that the evaluation of a is completed.

Let X_P be the iid random variable denoting the execution time of the expression P, which is characterised by the pdf $f_P(t)$. Then the random variable for the execution time of a **sequence** $P \cdot Q$ is the sum of the random variables denoting their execution times:

$$X_{P \cdot Q} = X_P + X_Q$$

Because X_P and X_Q are iid, their characterising pdfs can be convoluted to get the pdf characterising the execution time of the sequence [Tri01, RFB04]:

$$f_{P \cdot Q}(t) = (f_P \circledast f_Q)(t),$$

An **alternative** expression $P +_\pi Q$ models that either P is executed with probability π or Q is executed with probability $1-\pi$. Let $u = u(\cdot)$ be a sample from the pdf of the uniform distribution $u(x)$ between zero and one. Then the random variable for the execution time of the alternative expression is given by [Tri01, RFB04]:

$$X_{P +_\pi Q} = \begin{cases} X_P, & \text{if } 0 \leqslant u < \pi \\ X_Q, & \text{if } \pi \leqslant u \leqslant 1 \end{cases} .$$

The pdf of an alternative expression is the weighted sum of the single pdfs:

$$f_{P +_\pi Q}(t) = \pi f_P(t) + (1 - \pi) f_Q(t)$$

A **loop** expression $P^{*(l)}$ models that P is executed i times in a row according to the pmf $l(i) = P_l(X = i)$. Let $F_l(x)$ be the cdf of $l(i)$ and $u = u(\cdot)$ again be a

sample from the pdf of the uniform distribution $u(x)$ between zero and one. Then the random variable for the execution time of the loop expression is given by [KF06]:

$$X_{P*^{(l)}} = \begin{cases} 0 & 0 \leqslant u < F_l(0) \\ X_{P,1} & F_l(0) \leqslant u < F_l(1) \\ X_{P,1} + X_{P,2} & F_l(1) \leqslant u < F_l(2) \\ & \vdots \\ X_{P,1} + X_{P,2} + \ldots + X_{P,N} & F_l(N-1) \leqslant u < F_l(N) \end{cases}$$

with $N \in \mathbb{N}_0$ the last value with $l(N) > 0$ and $l(j) = 0 \quad \forall j \in \mathbb{N}_0, j > N$. $X_{P,i}$ is the ith instance of random variable X_P. The pdf of a loop expression is the weighted sum over a number of sequences:

$$f_{P*^{(l)}}(t) = \sum_{i=0}^{N} l(i) \left(\overset{i}{\underset{j=1}{\circledast}} f_P \right)(t)$$

As an example for the computations, consider the SRE $P = (a +_{0.4} b) \cdot c^l$, which involves a sequence, an alternative, and a loop. Let $l(1) = 0.1$ and $l(2) = 0.9$, so that the number of loop iterations for executing c is bounded to a maximum of 2. The pdf of the overall sojourn time is given by the pdf

$$f_P = (0.4 * f_a(t) + 0.6 * f_b(t)) \circledast \sum_{i=0}^{2} l(i) \left(\overset{i}{\underset{j=1}{\circledast}} f_c \right)(t)$$

To implement this solution technique, the involved continuous pdfs need to be approximated with discrete pmfs. The `SamplePDF` described in Chapter 3.3.5 provides a solution for this. The implementation of the solver developed in the context of this thesis (`SRE-Solver`) builds the binary abstract syntax tree of a given SRE, where the leafs represent terminal symbols and all inner nodes represent the operations sequence, alternative, and loop.

The `SRE-Solver` traverses the abstract syntax tree bottom-up with a visitor and performs above's operations for sequence, alternative, and loop on each inner node by using the children's pdfs as operands. Once the visitor reaches the root node, it has determined the overall sojourn time of the whole SRE.

The `SRE-Solver` uses `SamplePDFs` to approximate the involved pdfs. For their convolution, it uses Fast Fourier Transformation, which reduces the computational complexity to $O(NlogN)$, where N is the number of sampling points used [FBH05].

6.3.5 Mapping from PCM Instance to StoRegEx

For model transformation and implementing the `SRE-Solver`, there is an Ecore implementation of the SRE meta-model (Fig. 6.7). The mapping from a PCM instance with computed contexts to an SRE instance Fig. 6.8 requires a number of restrictions on the PCM instance to be applicable, as the SREs cannot express several concepts of the PCM. Branches may contain only two branch conditions. The PCM instance must not include forks, passive resource locking (acquisition/release), or multiple resource demands on a single internal action. Furthermore, the mapping ignores scheduling policies of processing resources, as SREs only support non-concurrent single user scenario, where resource queues requiring scheduling are unnecessary.

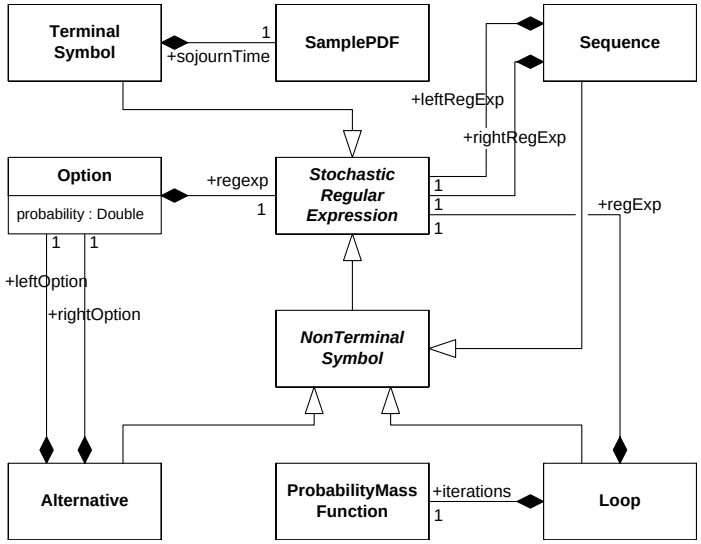


Figure 6.7: Stochastic Regular Expressions in Ecore (meta-model)

The mapping processes both usage scenarios and RDSEFFs. Fig. 6.8 only depicts the mapping from RDSEFF instances, as the mapping for usage scenarios is very similar. The mapping directly copies branch probabilities and iteration number from the `ComputedUsageContexts` to the SRE instance. Resource demands from the `ComputedAllocationContexts` become the sojourn times of SRE terminal symbols. No further conversion is needed.

The mapping is implemented in Java using a visitor for usage scenarios and a visitor for RDSEFFs. It uses the context wrapper to access information from the computed contexts. The visitors build up an SRE instance while traversing the model. It is then passed to the `SRE-Solver`, which is described in the previous section.

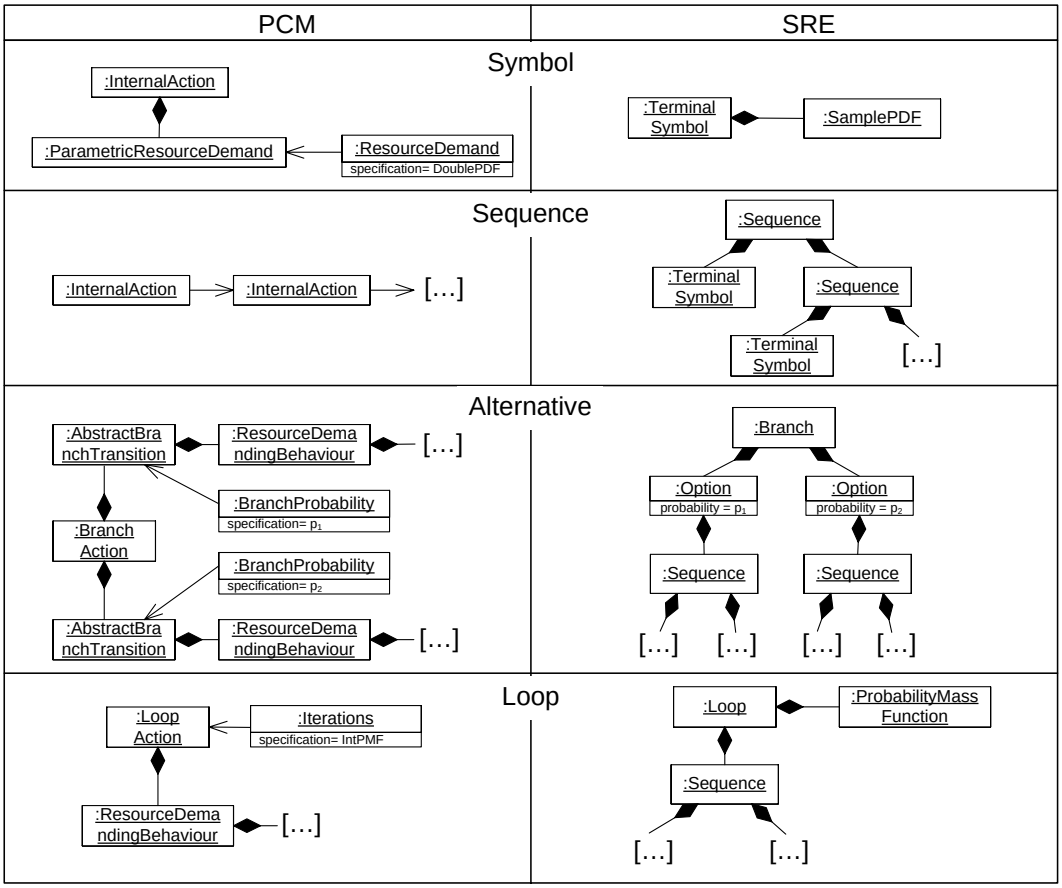


Figure 6.8: Mapping from PCM instance to SRE

6.3.6 Assumptions

The SRE model is a rather restricted performance model and does not exploit the full expressiveness of the PCM. Several assumptions underlay this model, which may limit its applicability in certain situations:

No concurrency: The model is not capable to express any kind of concurrent behaviour. It does not allow neither control flow forks of single requests, nor resource contention or passive resource possession for multiple requests. This is the most severe restriction of the model, and it has several implications. Performance analysts can apply SREs only for single-user scenarios, whereas in practice the vast majority of performance-critical use cases are multi-user scenarios, where resource contention can lead to the violation of performance goals.

However, the model is capable of making a first, precise prediction for single-user scenarios. The missing contention overheads simplify the model to a great extent and enable a straight-forward model solution even for arbitrarily distributed sojourn times. Other than a simulation that uses sampling and repeated execution to approximate the distribution functions, the `SRE-Solver` uses the complete distribution function for its computations. Therefore, a simulation may exclude certain improbable outliers in the distribution functions, which would only occur during very long simulation runs. The `SRE-Solver` includes such outliers in the prediction making it more accurate.

The single-user scenario solution for SRE models is related to the software execution model proposed by Smith et al. [Smi02]. They use a similar behavioural description, but only use constant values for execution times and loop iteration numbers. Their model does support control flow forks, but assumes a single available processor for each thread (i.e. no contention) and no influence to the execution time due to memory accesses. Smith et al. point out that their model is useful to study the initial feasibility of a design, before using more advanced solvers, which support resource contention. SREs should be used in the same manner.

Independent Random Variables The random variables used to specify the sojourn times in SREs need to be independent and identically distributed to enable convolution of the pdfs. This assumption might not hold in some realistic cases. For example, if two subsequent specified execution times model operations by the same resource, these times can be stochastically dependent. For example, if the resource is in overload mode, both random variables would be influenced.

Furthermore, the SRE model does not allow execution times to change over time. This might occur, if a component or system has a warm-up phase leading to slower execution times, which is followed by normal operation phase leading to normal execution times. However, for a quick feasibility study of an initial design this assumption might be negligible.

Markov property The embedded DTMC in the SRE model has the so-called Markov property, i.e., the probability of going from state i to state j in the next step is independent of the path to state i. This property simplifies analytical methods, but it might not hold in realistic use cases. The SRE model weakens this assumption for loops, because it does not allow cycles in the Markov chain. Therefore, inside a loop, the probability of going from state i to state j is not independent of the path to i, but it depends on the number of already executed iterations. This can lead to more accurate predictions (cf. Chapter 7.3).

However, the Markov property is still present in SREs for branches. The model assumes that previous analysis methods, such as the `Dependency-Solver`, have evaluated parameter dependencies on branch probabilities while incorporating violation to the Markov property and adjusting the model accordingly.

6.4 Transformation to Layered Queueing Networks

6.4.1 Overview

The Layered Queueing Network (LQN) model is a performance model in the class of extended queueing networks. It is a popular model with widespread use [BDIS04]. Like the PCM, it specifically targets analysing the performance of distributed systems. While ordinary queueing networks model software structures only implicitly via resource demands to service centers, LQNs model a system as a layered hierarchy of interacting software entities, which produce demands for the underlying physical resources such as CPUs or hard disks. Therefore, LQNs reflect the structure of distributed systems more naturally than ordinary queueing networks. In particular, they model the routing of jobs in the network more realistically.

In the context of this work, a model transformation from PCM instances (with computed context models) to LQNs has been implemented. The transformation offers at least two advantages: First, it enables comparing the concepts of the PCM with concepts of LQNs, which can be considered as a state-of-the-art performance

model. Second, the transformation makes the sophisticated analytical solvers and simulation tools for LQNs available to the PCM. Other than SREs, LQNs support concurrent behaviour, different kinds of workloads, asynchronous interactions, and different scheduling strategies. Therefore, it is possible to derive performance metrics such as resource utilizations and throughput from PCM instances, which is not possible with SREs. However, LQNs are restricted to exponential distributions and mean-values analysis as discussed later.

This section will first provide some background about LQNs and their development in recent years (Chapter 6.4.2). Then, it will describe the syntax and (informal) semantics of LQNs using the LQN meta-model and several examples (Chapter 6.4.3). Chapter 6.4.4 briefly describes two performance solvers for LQNs, before Chapter 6.4.5 presents the mapping from PCM instances to LQN instances. Finally, Chapter 6.4.6 compares the PCM model with the LQN model, as well as the existing PCM solvers with two available LQN solvers.

6.4.2 Background

LQNs have been developed by the Real-Time and Distributed Systems Group at Carleton University, Ottawa, for more than 20 years. The first papers by Woodside et al. [Woo84, WNHM86] about the "active server model" included in LQNs date back to 1984. The term Stochastic Rendezvous Network (SRVN) was introduced in 1989 [Woo89] to differentiate the model from queueing networks with flat resource modelling. In 1991, Petriu et al. [PW91] introduced a new solution method for SRVN called Task-Driven Aggregation (TDA), which was based on decomposition of the underlying Markov model. An efficient, heuristic solver (The Method of Layers, MOL) for hierarchically layered systems was introduced by Rolia et al. [RS95] in 1995, where also the term Layered Queueing was first coined. Meanwhile, the SRVN model was extended with multiple entries, phases, and requests that skip certain layers [WNPM95, NWPM95]. Franks et al. [FMN+96] combined the MOL and SRVN to create Layered Queueing Networks in 1996 and later added parallel operations within tasks [FW98].

LQNs have been applied for example to web servers [DFJR97], distributed data base systems [SW97], telecommunication systems [SPJN98], network routers [MW00], Enterprise Java systems [XOWM06], systems with replication [OFWP05], peer-to-peer systems [WWL04], and a distributed gaming platform [VDTD07]. Recent research focused on the derivation of LQNs from UML models [PS02] and Use

Case Maps [PW02], as well as the design of an intermediate language [PW04] to simplify transformation between UML models and LQNs or other performance models. Extensions to LQNs to specifically support component-based systems have already been discussed in Chapter 2.5.1.

6.4.3 Syntax and Semantics

This subsection describes the syntax and informal semantics of LQNs and provides a small example. Fig. 6.9 depicts the Ecore LQN meta-model generated using EMF [BSM+03] from the LQN XML schema definition available at [Rea]. The main elements of LQNs are processor, task, entity, activity, and precedence as explained in the following:

Processor An LQN consists of a number of `ProcessorTypes`, which represent physical resources, such as processors and hard disks. LQN activities (described below) use processors to consume time. A processor cannot issue requests to other entities, therefore it is only a server and cannot be a client.

Each processor has a single request queue with a particular `SchedulingType`. The LQN solvers support several scheduling disciplines: first-come first-serve, processor sharing, priority preemptive resume, random, head-of-line, processor sharing head-of-line, and processor sharing priority preemptive resume (cf. Appendix B.3 for more details). The `quantum` attribute of `ProcessorType` is needed only for the LQN simulation solver, which approximates processor sharing scheduling with round-robin scheduling and therefore needs this time slice specification.

It is possible to specify an attribute `multiplicity` for a processor. It models the number of service centers available for the `ProcessorType's` single request queue. On the other hand processors can have an attribute `replication`. In this case, besides the service centers, also the number of queues is multiplied. Both attributes may be combined to specify replication of multi-core processors.

Finally, performance modelers can use the `speedFactor` attribute to model a constant speed-up or slow-down of the processor. All demands by activities are then multiplied by this factor. This enables investigating the impact of introducing a faster CPU or hard disk into the system.

Task Each `ProcessorType` may contain a number of `TaskTypes`. Tasks are the main modelling elements in LQNs and can be used to represent various real-

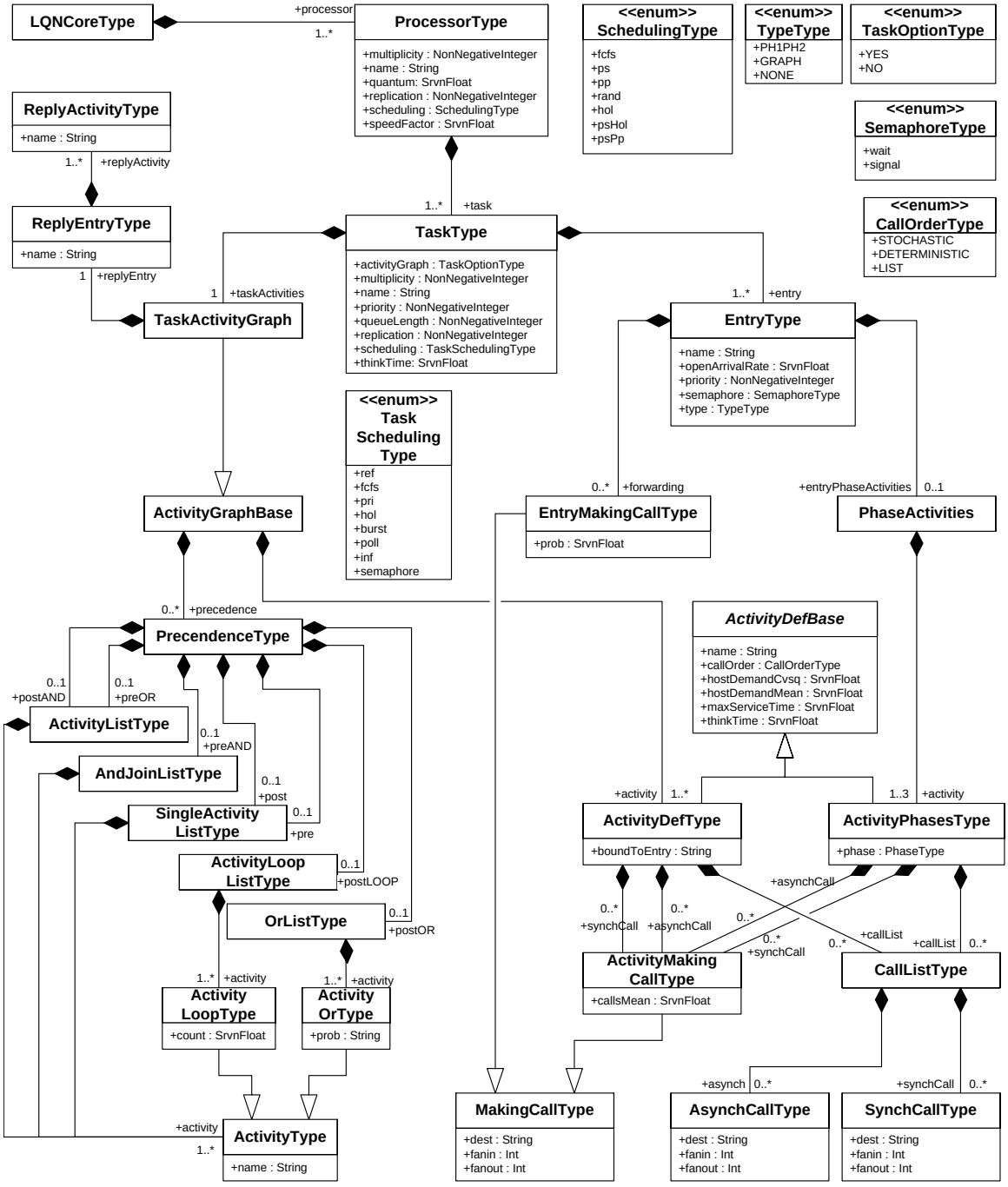


Figure 6.9: LQN meta-model (generated from XML-Schema)

life entities. This includes customers, software servers, software components, services of hardware resources, passive resources such as buffers and semaphores, or databases. Tasks can call other tasks and be invoked from other tasks, i.e., they can act both as a client and as a server. Therefore, tasks were formerly also known as "active server". The term 'task' has been taken from the Ada programming language, where it represents a process.

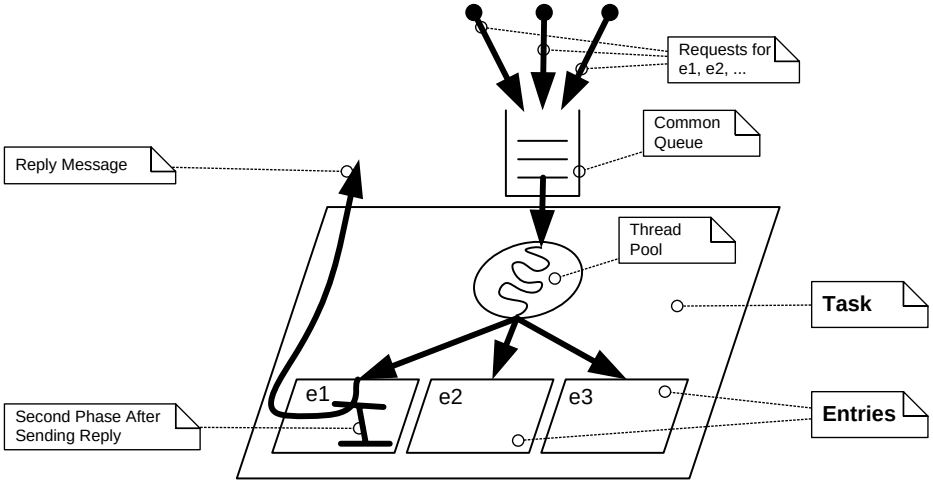


Figure 6.10: LQN-Tasks Illustration [Woo02]

Fig. 6.10 illustrates the basic structure of a task. The notation is similar to Use Case Maps [BC96], i.e., thick black arrows indicate the control flow of requests through the task. A task consists of a number of entries (`EntryType`) and has a single request queue. An entry is the starting point for a service the task provides. Requests to the task are added to the task's request queue, which is either unlimited or has a limited length specified by the attribute `queue-length`. The task serves the request according the `TaskSchedulingType`. Supported scheduling disciplines are first-come first-serve, priority preemptive resume, head-of-line, burst, poll, and infinite server (cf. Appendix B.3 for more details).

Usually, a task serves only a single request at a time. But, setting the `multiplicity` attribute higher than 1 is interpreted as the task having a thread pool with the specified capacity. This enables serving multiple requests in parallel. Furthermore, with the `replication` attribute it is possible to make copies of the request queue. In this case, requests get assigned to the different queues in a round-robin fashion. Once a request has obtained a thread instance, it executes the activities within the requested entry. The thread can send a reply message back to

the client and then enter a second phase of asynchronous execution, which does not reply to the client (details follow below).

During the execution of an entry, its included activities may call other tasks. The set of tasks in an LQN is structured into layers. A task in a particular layer may only request service from tasks in lower layers, but not upper layers. However, it is possible to send reply messages to upper layers. This constraint makes the graph of tasks acyclic and avoids deadlocks among requests [Woo02]. Furthermore, the layering is not strict, i.e., tasks may skip layers and for example directly call tasks at the lowest layer.

A task can represent a customer of the system. In this case, the task only serves as a client issuing requests, but not as a server receiving requests. Such tasks are called ”reference tasks” and are specified by setting the `TaskSchedulingType` to 'ref'. The attribute `multiplicity` then models the user population and the attribute `thinkTime` models the delay each user waits before again issuing a request after returning from former requests. This reflects the modelling of a closed workload in an ordinary queueing network. Performance modelers can specify open workloads by using the `openArrivalRate` attribute of the class `EntryType` (cf. Fig. 6.9).

A task can also represent a semaphore to generically model passive resources. In this case, the semaphore task always contains two entries ”SIGNAL” and ”WAIT” and has the `TaskSchedulingType semaphore`. Other tasks can call the ”SIGNAL” entry with synchronous or asynchronous requests. The ”WAIT” entry, however, may only be called with synchronous requests. Once the ”WAIT” entry has accepted a request, it accepts no further requests until the ”SIGNAL” entry processes a request. It is possible to invoke both entries from different tasks. Furthermore, using the `multiplicity` attribute counting semaphores can be realised.

The sequence of activities executed by entries of a task are usually specified using `EntryActivityGraphs`, so that each entry has its own activity graph. However, `EntryActivityGraphs` are not supported by the current version of the available solvers. Instead, they only support `TaskActivityGraphs` (cf. Fig. 6.9), which specify a single activity graph for a whole task. These `TaskActivityGraphs` have been used for the model transformation from PCM instances to LQNs as explained in Chapter 6.4.5.

Entry An `EntryType` represents the starting point for a service provided by a task. For example, it models services provided by a web server or services of a hardware resource (e.g., read/write for a hard disk). Entries accept either only syn-

chronous or asynchronous requests, but not both. In the case of synchronous requests, an entry either generates a reply itself, which it transfers back to the calling client, or forwards the request to another task using an `EntryMakingCallType`, which then in turn generates the reply for the calling client.

If an entry synchronously serves a request by itself, its behaviour consists of several phases (Fig. 6.11). After a task has received a request from a client and selected it from the request queue, an entry serves the client, i.e., it consumes time on the underlying resources. During this, the client blocks and waits for a reply. Upon finishing the first phase, the entry sends a reply back to the client, so it can continue execution. In parallel, an entry can then perform an arbitrary number of additional phases, which include calls to other tasks or time consumption of resources. These additional phases are performed autonomously from the client.

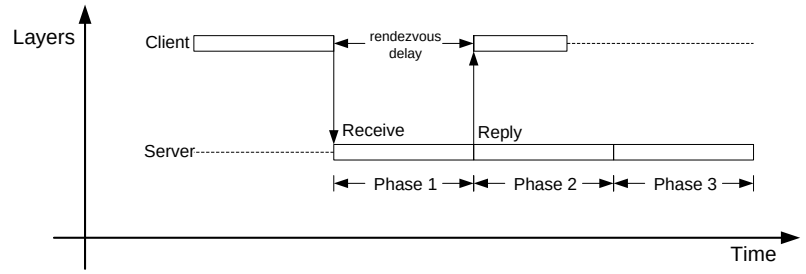


Figure 6.11: Phases in LQNs Illustration [FMW+07]

The phases are intended to model common behaviour in distributed systems, where servers return the control back to the client as early as possible to increase the responsiveness of the system. For example, a database commit is well modelled with phases, as it returns the control back to the user if the operation is possible, and then performs the operation asynchronously from the client in the background [Woo02]. Notice that the current LQN solvers restrict the number of phases to a maximum of 3.

The behaviour of an entry (or task), i.e., the sequence of execution steps, can be specified either via `PhaseActivities` (attribute `type=PH1PH2`), an `EntryActivityGraph` (attribute `type=GRAPH`) or an `TaskActivityGraph` (attribute `type=NONE`). A `PhaseActivity` is a short hand notation for the phases above (Fig. 6.11) and models a sequential execution of a single activity for each of the up to three phases. The graphs, which are either attached to entries or tasks, model the control flow using sequences, alternatives, loops, and forks with `PrecedenceTypes`.

Activity An activity (`ActivityDefBase`) consumes time on a processor or calls other tasks. For modelling time consumption, activities either specify a mean service time (attribute `hostDemandMean`) or the coefficient of variation for the service time (attribute `hostDemandCvsq`), which is the variance of the service time divided by the square of the mean. LQNs assume that the service time is exponentially distributed.

For modelling calls to other tasks, activities contain `ActivityMakingCallTypes` for single requests or `CallListTypes` for multiple requests. The call order of these requests may either be deterministic or stochastic (attribute `callOrder`). A deterministic call order implies issuing the exact number of specified requests to the underlying tasks. A stochastic call order implies issuing a random number of requests to the underlying tasks with the specified mean value (attribute `callsMean`). LQNs assume a geometrically distributed number of stochastic requests.

For an synchronous call an entry must generate exactly one reply, so that the client can continue execution. When using `TaskActivityGraphs` as done here, `ReplyActivityType` can be used to explicitly declare an activity inside a graph as the reply activity. Its `name` attribute is then the same as the `name` attribute the reply activity (i.e., the reference is established via string matching). When using `PhaseActivities`, the reply is implicitly generated after the first phase activity completes.

Precedence `PrecedenceTypes` connect activities to each other in sequences, branches, loops, or forks to form a control flow graph. They are split into 'pre'-precedences to join or merge activities and 'post'-precedences to branch or fork activities.

Using a 'pre'-precedence, it can be specified that an activity follows exactly one other activity in a sequence (`SingleListActivityType`), that an activity waits for all previous activities before continuing execution (`AndJoinListType`), or that an activity waits for only a single of all previous activities (`ActivityListType`).

Using a 'post'-precedence, it can be specified that an activity is followed by exactly one other activity (`SingleListActivityType`), by one of a list of activities with a given probability (`OrListType`), by all of a list of activities (`ActivityListType`), or by a repetition of the following activities (`ActivityLoopListType`). It is assumed that the number of loop iterations follows an geometrical distribution with the mean value given by the attribute `count`. It is not possible to have replies from activities inside a loop, because the number of iterations is random.

Example Fig. 6.12 shows an LQN example instance using the common concrete graphical syntax for LQNs to illustrate the concepts explained above. The figure shows tasks as large parallelograms and processors as circles. Rectangles within entries represent activities. The replication of tasks or processors is indicated via multiple parallelograms or circles on top of each other.

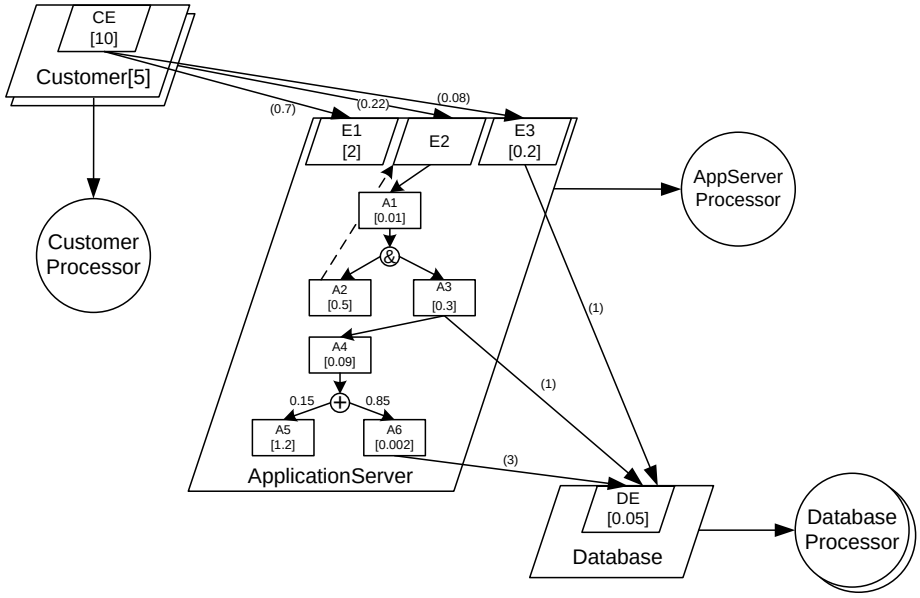


Figure 6.12: Simple LQN Example with 3 Layers

The example models a simple three tier architecture with a client layer, an application server layer, and a data base layer. In this example, each layer has only a single task. The Customer task runs on the Customer processor and is replicated 5 times, therefore modelling a closed workload with a user population of 5. The think time is 10 seconds given in the CE-entry in square brackets. Clients call three different entries of the Application Server task running on the AppServer Processor with the given probabilities.

Its entries E1 and E3 are specified using phase activities, while entry E2 is specified with an entry activity graph. E1 consumes 2 time units on the AppServer Processor and then returns to the client without calls to other tasks. E2 includes an activity graph consisting of several activities connected via OR or AND precedences. A2 is the return activity of the entry and sends a reply message back to the client. The activities A3 and A6 both consume time on the AppServer Processor and issue requests to the Database task. Additionally, the entry E3 requests service from the Database task.

Each call of the DE-entry of the Database task takes 5 time units. The Database executes on a replicated Database Processor, i.e., a multiprocessor system. The structure of this LQN is strictly hierarchical, lower layers do not issue requests to upper layers.

6.4.4 Solution Techniques

Multiple solution techniques have been developed to derive performance metrics, such as response times, throughput, and resource utilisation, from LQNs, as described in Chapter 6.4.2. Currently, the LQN tool suite contains two different solvers: the analytical solver LQNS and the simulation tool LQSIM. These solvers are the result of combining some of the formerly developed techniques (e.g., MOL, TDA, cf. Chapter 6.4.2). Both solvers use the same input format and produce similar outputs in human-readable text or XML. The outputs contain for example mean service time per task or activity, service time variance per task or activity, throughput per task, and utilisation per processor, task, and phase.

LQNS LQNS [Fra99] is an analytical solver combining the strength of the SRVN solver [WNPM95] and the MOL solver [RS95]. It decomposes an LQN into several separate queueing networks and solves these sub-models individually using mean value analysis (MVA) [RL80].

MVA is a popular and efficient solution algorithm for product-form queueing networks. This is a very restricted class of queueing networks, which for example requires exponentially distributed service times, a fixed user population, and service rates only dependent on the number of customers at a service center. Features such as simultaneous resource possession or fork/join interaction violate these assumptions. The former makes the service rates of different service centers dependent on each other, while the latter changes the user population in the net.

SRVN and MOL provide heuristic solutions to analyse queueing networks that include these advanced features by decomposing them into simpler sub-models and then performing approximate instead of exact MVA on them. MOL in particular uses the Linearizer algorithm [CN82] to estimate the queue lengths in the sub-models. The results of these heuristics are in many cases sufficiently accurate [Fra99].

The solvers then use the MVA results from each sub-model to adapt the MVA parameters of other sub-models they are connected to. Afterwards, MVA is car-

ried out again in an iterative process. The process either stops when a user-defined maximum number of iterations has been reached or when the results converge to a user-defined interval.

The SRVN solver and the MOL solver supported several different features, which later have been combined in LQNS. For example, the MOL solver supported the processor sharing scheduling discipline and was able to analyse multi-servers, but was restricted to closed workloads. Instead, the SRVN solver included open workloads, but was restricted to single servers and did not support processor sharing. LQNS supports all of these features as detailed in [Fra99].

LQSIM LQSIM is a discrete-event simulation framework for LQNs. It was formerly called ParaSRVN and uses the ParaSol simulation environment [Nei91]. As a simulation approach, it imposes the fewest restrictions on the input models, but usually takes longer to execute than the analytical solvers.

LQSIM creates simulations objects from tasks, processors, and queues of an LQN using a library provided by ParaSol. It creates lightweight threads for requests and uses them to simulate the execution of the modelled system. During execution, LQSIM collects statistics on throughput and delays of each thread. Once it reaches a user-specified confidence interval, it stops the simulation and prints out the results for several performance metrics, such as response times, throughput, and resource utilisation. Other than LQNS, LQNSIM can additionally provide a service time distribution including a histogram in text form for a specific entry in its results.

6.4.5 Mapping from PCM instances to LQN

The transformation PCM2LQN maps PCM instances with computed contexts (Chapter 6.2) to LQN instances. The resulting LQNs are valid input for the solvers LQNS and LQSIM. This subsection describes the rationale behind the mapping, the actual transformation for PCM elements to LQN elements, and the technical realisation of the mapping.

Rationale There are at least two alternatives for mapping from PCM instances to LQNs (Fig. 6.13). The first alternative is to resolve the parameter dependencies in a PCM instance using the Dependency Solver (Chapter 6.2) and then map the resulting PCM instance including the computed contexts to an LQN instance. The second alternative is to map a PCM instance including parameter dependencies to an

instance of the Component-Based Modelling Language (CBML), which is an extension for LQNs to support modelling software components [WMW03, WW04] (also see Chapter 2.5.1). The resulting CBML instances can then be assembled into an LQN instance.

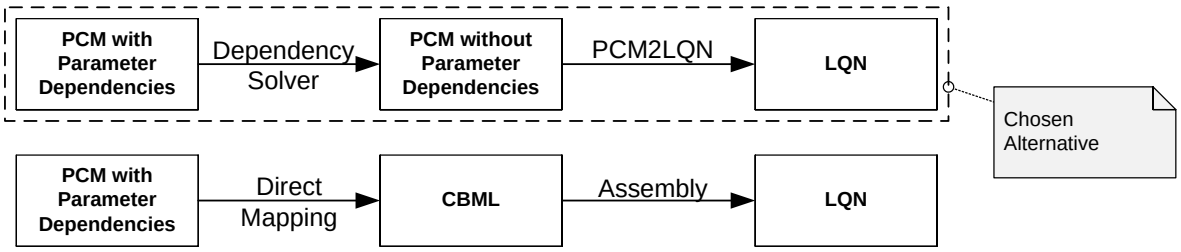


Figure 6.13: Two Alternatives for a PCM to LQN Mapping

In this work, the first alternative has been chosen for the following reasons. While CBML also features a parameterisation, it only supports global parameters similar to the concept of component parameters introduced in Chapter 4.1.3. The values of these parameters can only be specified as strings. Therefore, it is not appropriate to map the PCM stochastic expressions to them. Furthermore, the CBML parameterisation does not support parameterising branch conditions or resource demands, which may occur in PCM instances. Therefore, a mapping to CBML would only be partial in terms of parameter dependencies.

PCM2LQN uses a PCM instance with computed contexts as input and thus does not preserve parameter dependencies in the LQN mapping. PCM2LQN does not map to certain LQN elements, because they are not supported by the available LQN solvers. For example, the solvers do not support `EntryActivityGraphs`, which would be the natural choice for mapping RDSEFFs. Instead, PCM2LQN uses `Task-ActivityGraphs` for mapping RDSEFFs.

Mapping PCM Resources PCM2LQN transforms PCM elements as follows. `BasicComponents`, `CompositeComponents`, and `ResourceContainers` are not reflected in the LQN resulting from the models. They are merely container classes for RDSEFFs and `ProcessingResourceSpecifications` and do not have counterparts in the performance domain.

The basis for the resulting LQN are processors, which form the leafs of the resulting LQN acyclic graph or tree. Thus, each `ProcessingResourceSpecification` in a PCM instance is transformed

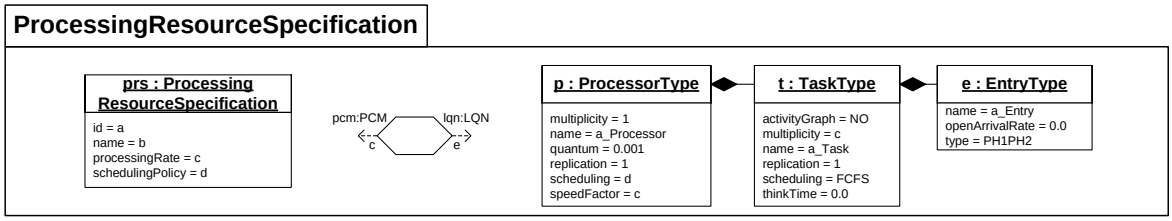


Figure 6.14: Mapping PCM2LQN: `ProcessingResourceSpecification`

into an LQN processor with a multiplicity of 1 (Fig. 6.14). Its `processing-rate` directly translates to the processor's `speed-factor`. PCM2LQN can directly map the FCFS and processor sharing scheduling disciplines in the PCM to the respective LQN `SchedulingTypes`. However, infinite server scheduling (called `DELAY`) in the PCM cannot be mapped, as their is no counterpart in `SchedulingTypes`. For using the processor sharing scheduling discipline, the LQSIM solver needs a `quantum` specification, which is not available in the PCM. PCM2LQN sets it to 0.001 by default, because LQSIM expects a value greater than zero.

PCM2LQN also generates a task with a dummy entry for each processor. This is necessary, because an LQN model is invalid for the LQN solvers if a processor is never used by any task. After transforming all `ProcessingResource-Specifications` to LQN processors, PCM2LQN processes PCM usage models.

Mapping PCM Workloads A `ClosedWorkload` becomes an LQN reference task with a `multiplicity` equaling the PCM user population (Fig. 6.15). The think time of the PCM usage model gets the `thinkTime` of the task. Additionally, PCM2LQN creates an LQN entry for the PCM `ScenarioBehaviour` corresponding to this workload. The user actions in the `ScenarioBehaviour` are mapped to LQN activities in a `TaskActivityGraph` connected to this task. Furthermore, PCM2LQN generates a dummy processor for the created reference task, because each task must run on a processor. In this case, the activities created for PCM user actions, which do not request processing from resources, will not request any processing from this processor.

Each `OpenWorkload` becomes an LQN task with FCFS scheduling and a think time of zero (Fig. 6.16). The scheduling discipline is irrelevant as the task will not receive but only generate requests. Like for the closed workload, PCM2LQN creates an LQN entry with an associated `TaskActivityGraph` to represent the `ScenarioBehaviour`. The inter-arrival time of the open workload gets mapped

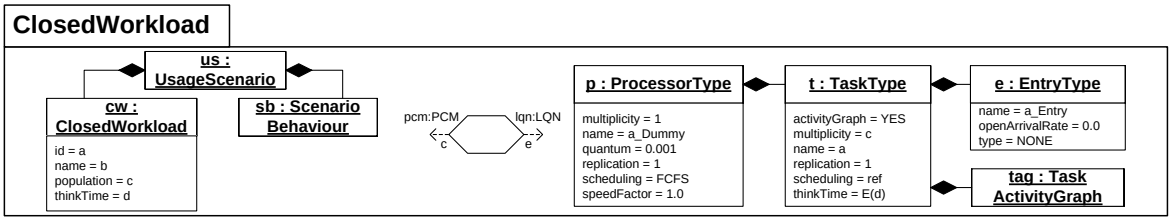


Figure 6.15: Mapping PCM2LQN: `ClosedWorkload`

to the `openArrivalRate` of this LQN entry. The rate is the reciprocal value of the inter-arrival time's expected value. As for the closed workload, PCM2LQN generates a dummy processor for the task representing the open workload.

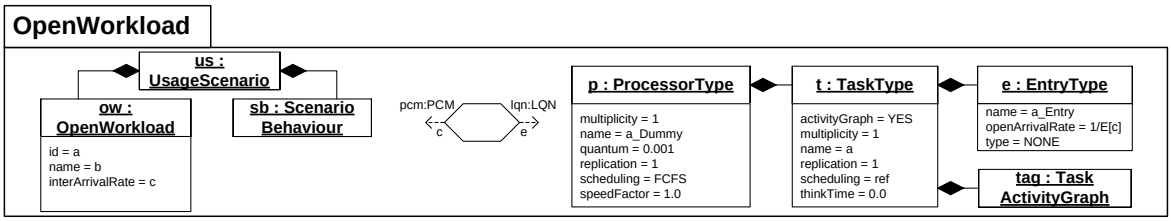


Figure 6.16: Mapping PCM2LQN: `OpenWorkload`

Mapping PCM User Actions `EntryLevelSystemCalls` in PCM instances model calls to the system. PCM2LQN creates an `ActivityDefType` with an attached synchronous `ActivityMakingCallType` for each of these entry calls (Fig. 6.17). The latter invokes the entry of the task representing the called RDSEFF. The former contains a zero `hostDemandMean` and a zero `thinkTime`, as it is assumed in the PCM that such calls do not consume any time. PCM2LQN adds the created activity to the `TaskActivityGraph` of the current `ScenarioBehaviour`. Additionally, it adds a `PrecedenceType` with two `SingleActivityListTypes` to the task activity graph to connect the activity with its successor.

PCM2LQN maps user `Delays` in PCM instances to LQN `ActivityDefTypes` (Fig. 6.18). The PCM user delay is a pdf modelling the waiting or thinking of a user. As the LQN only allows constant values for timing annotations, PCM2LQN calculates the expected value $E(c)$ of the user delay c and uses it as the `thinkTime` of the `ActivityDefType`. The activity modelling the user delay produces no additional demands to any resources (`hostDemandMean = 0.0`). PCM2LQN adds the activity to the `TaskActivityGraph` created for the current `Scenario-`

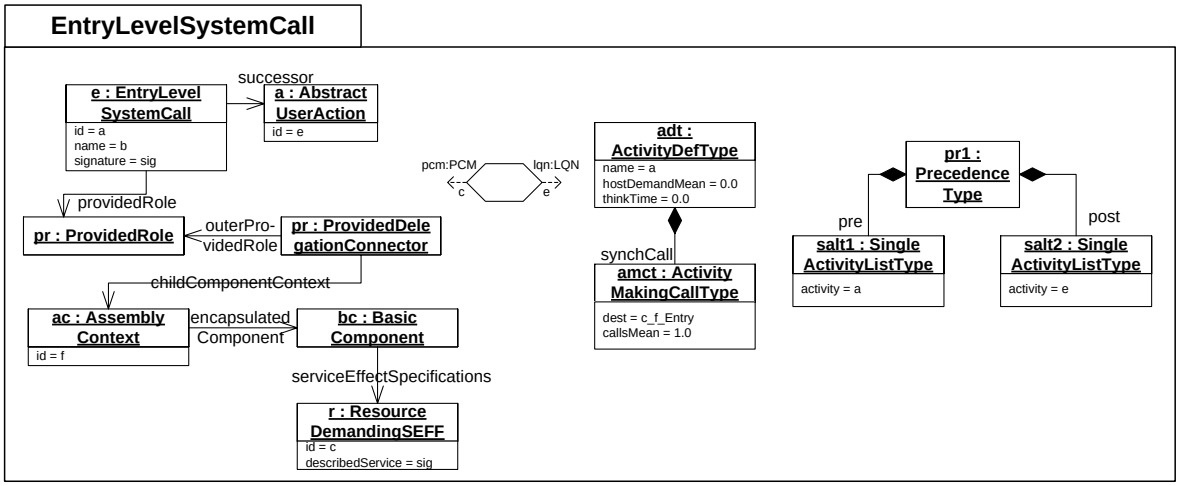


Figure 6.17: Mapping PCM2LQN: `EntryLevelSystemCall`

`Behaviour`. It also creates a `PrecedenceType` as explained before for the `Entry-LevelSystemCall` node to connect the activity to its successor.

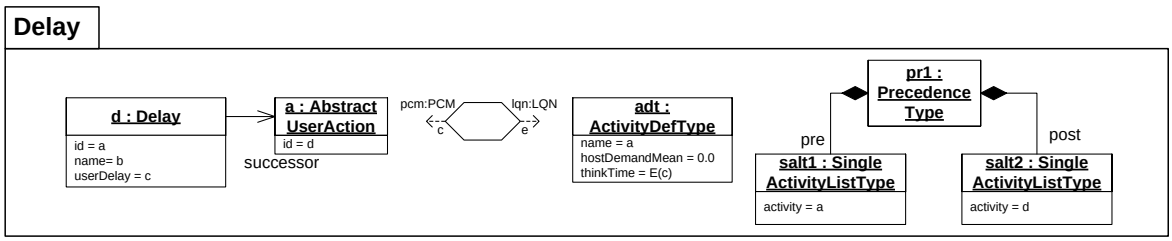


Figure 6.18: Mapping PCM2LQN: `Delay`

If the usage scenario contains a `Branch`, PCM2LQN transforms it into LQN elements as depicted in Fig. 6.19. It creates a single `ActivityDefType` with zero `hostDemandMean` and zero `thinkTime`, which reflects initiating the branch. This activity gets connected to the newly created `PrecedenceType` `pr1`. It includes a `OrTypeList`, which contains an `ActivityOrType` referencing the activity created later to represent the `Start` actions of the branched `Scenario-Behaviour` (`name=a`). This transformation is carried out for each `Branch-Transition`. The probabilities of a `BranchTransition` can directly be mapped to the `prob` attribute. PCM2LQN merges the branched control flows using a second `PrecedenceType` `pr2`. It includes a list of `ActivityListTypes`, which reference the `Stop` actions of the branched `ScenarioBehaviours` and connects them to the successor action of the `Branch`.

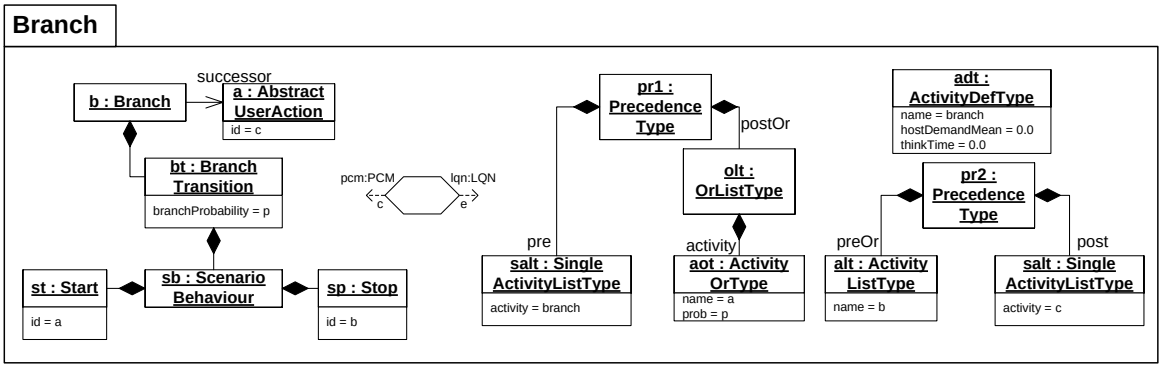


Figure 6.19: Mapping PCM2LQN: `Branch`

The mapping for `BranchActions` in RDSEFFs is almost the same as the mapping for `Branches` in usage models and will not be depicted here for brevity. Instead of directly mapping the branch probabilities of the `BranchTransitions`, PCM2LQN then accesses the computed usage contexts via the `ContextWrapper` (Chapter 6.2.4) to retrieve the probability resulting from solving parameter dependencies. Furthermore, the mapping adds the GUID of the current `AssemblyContext` to the `name` attributes of the created LQN elements. Otherwise, the mapping is the same.

If a usage scenario contains a loop, PCM2LQN creates a new task with an according processor, entry, and task activity graph for the loop body (Fig. 6.20). This is a workaround, because LQN `ActivityLoopTypes` do not support arbitrary behaviour inside loop bodies, but only a sequence of activities which are called repeatedly. As PCM instances support arbitrary behaviour inside loop bodies, the mapping creates a new task for a PCM loop body, which then contains a `TaskActivityGraph` allowing arbitrary behaviour.

To invoke the loop body, the mapping creates an `ActivityMakingCallType` in the current `TaskActivityGraph` that issues a number of synchronous calls to the newly created task according to the attribute `callsMean`. PCM2LQN uses the expected value $E(c)$ if the number of loop iterations in the PCM instance was a pmf c for this attribute. The execution of a loop does not create any additional resource demands or think times.

The mapping for `LoopActions` and `CollectionIteratorActions` from RDSEFFs is almost the same as the mapping of `Loops` of the usage model. In that case, PCM2LQN retrieves the number of loop iterations from the computed usage context via the `ContextWrapper` (Chapter 6.2.4) instead of directly using the at-

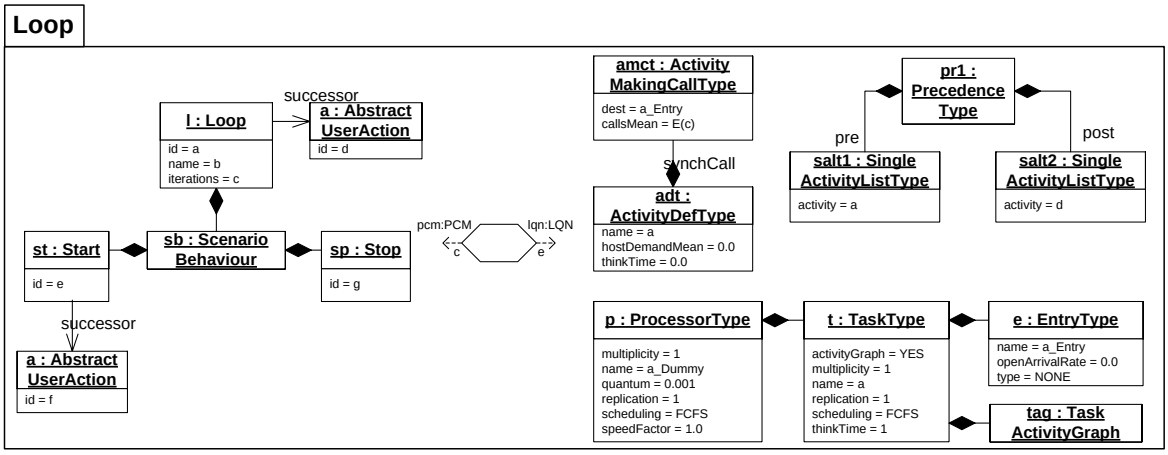


Figure 6.20: Mapping PCM2LQN: `Loop`

tribute `iterations`, and adds the GUID of the current `AssemblyContext` to the created LQN elements. Otherwise the mapping is the same, and therefore not depicted here.

Mapping PCM RDSEFFs After transforming an `EntryLevelSystemCall`, PCM2LQN maps the corresponding RDSEFF in the specific `AssemblyContext` to a new task with a corresponding processor, entry, and task activity graph for the included `ResourceDemandingBehaviour` (Fig. 6.21). Notice that for this mapping, all created LQN elements with a `name` attribute contain the `AssemblyContext` GUID f in their `name` value. If the same RDSEFF is used in another `AssemblyContext`, the corresponding LQN elements get created again with different `name` values.

The created processor for the RDSEFF is again a dummy processor to make the resulting LQN valid, but will not be used by the activities created for the actions inside the RDSEFF, as they use the formerly created processors from `ProcessingResourceSpecifications`. The `ActivityDefType` PCM2LQN creates for the `StartAction` of the RDSEFF's `ResourceDemandingBehaviour` gets connected to the entry of the newly created task (cf. attribute `boundToEntry`). PCM2LQN uses the activity created for the `StopAction` of the behaviour to map it to the `ReplyActivity`. The newly created `PrecedenceType` connects the `StartAction` to its successor action in a sequence.

PCM `InternalActions` may contain multiple `ParametricResourceDemands`. Multiple resource demands by an `InternalAction` are represented

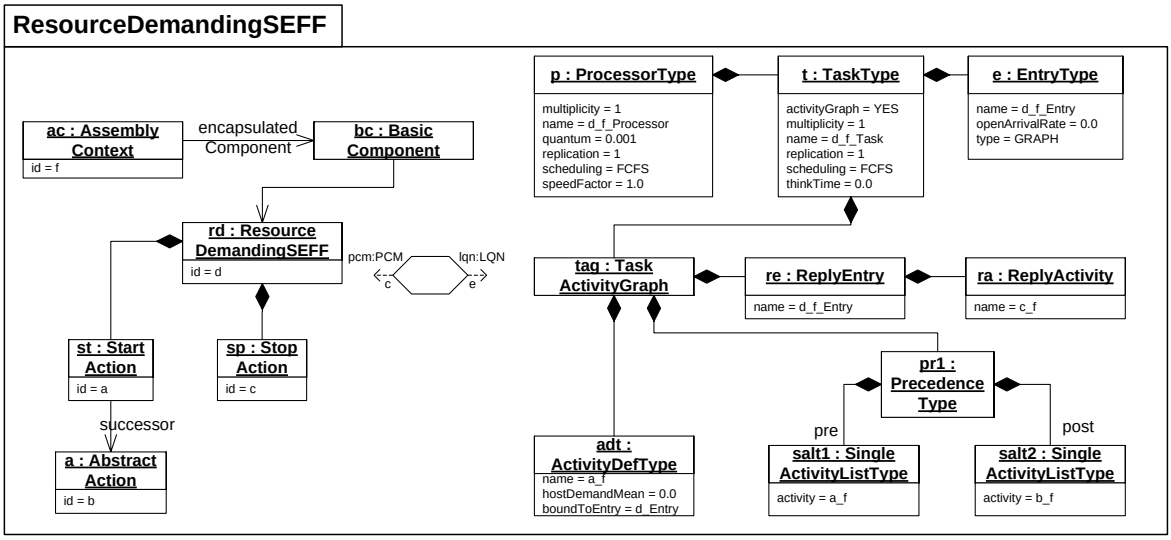


Figure 6.21: Mapping PCM2LQN: `ResourceDemandingSEFF`

by a sequence of activities in the LQN. PCM2LQN creates a new entry in the task created earlier for the corresponding `ProcessingResourceSpecification` (`id` = g) to express a `ResourceDemand` in the LQN (Fig. 6.22). This entry includes a `PhaseActivities` with a single `ActivityPhasesType` for the first phase. Further phases are not supported by the PCM. The phase's `hostDemandMean` attribute is assigned to the expected value $E(x)$ if the `ResourceDemand` specification in the RDSEFF was a pdf or pmf x. If x was a constant, the value is used directly. After creating the new entry to reflect the resource demand, PCM2LQN creates a synchronous `ActivityMakingCall` in the current RDSEFF to call this entry (`callsMean = 1.0`).

The mapping of forks in RDSEFFs resembles the former mapping of branches. So far, PCM2LQN only supports synchronous forks inside an RDSEFF. PCM2LQN creates an `ActivityDefType` for a `ForkAction` with zero `hostDemandMean` and zero `thinkTime`. The `StartAction` of each synchronously forked `ForkedBehaviour` then gets inserted to an `ActivityListType` modelling the successor activities (`postAND`) of the activity created for the fork. The `StopAction` of each `ForkedBehaviour` gets inserted into an `AndJoinListType`, which models waiting for all forked threads to finish before continuing execution. The successor activity of this list is the activity created for the successor of the `ForkAction` in the RDSEFF.

RDSEFFs may also include `AcquireActions` and `ReleaseActions` to model

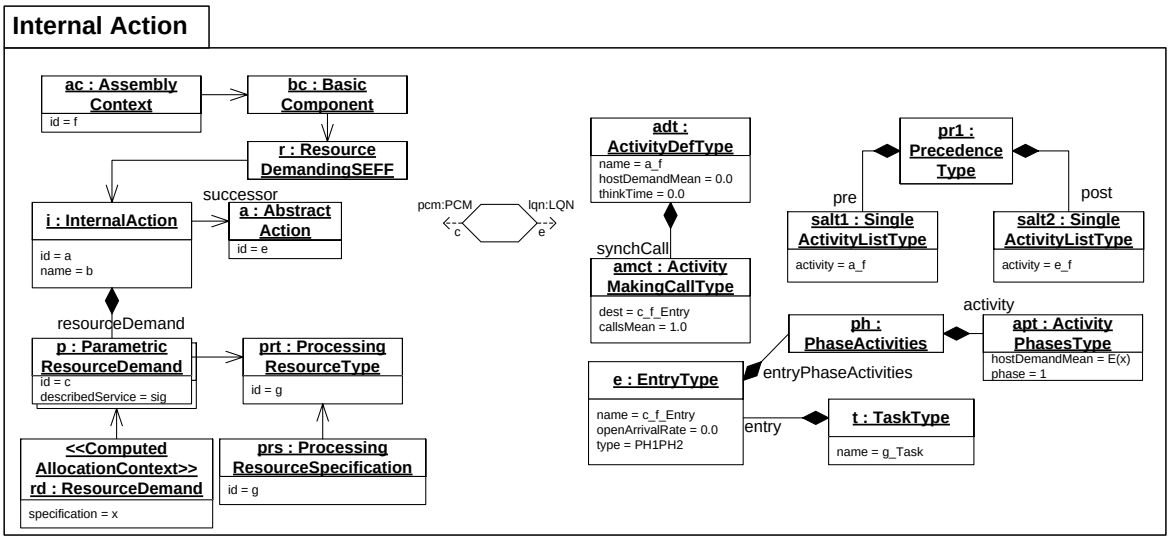


Figure 6.22: Mapping PCM2LQN: `InternalAction`

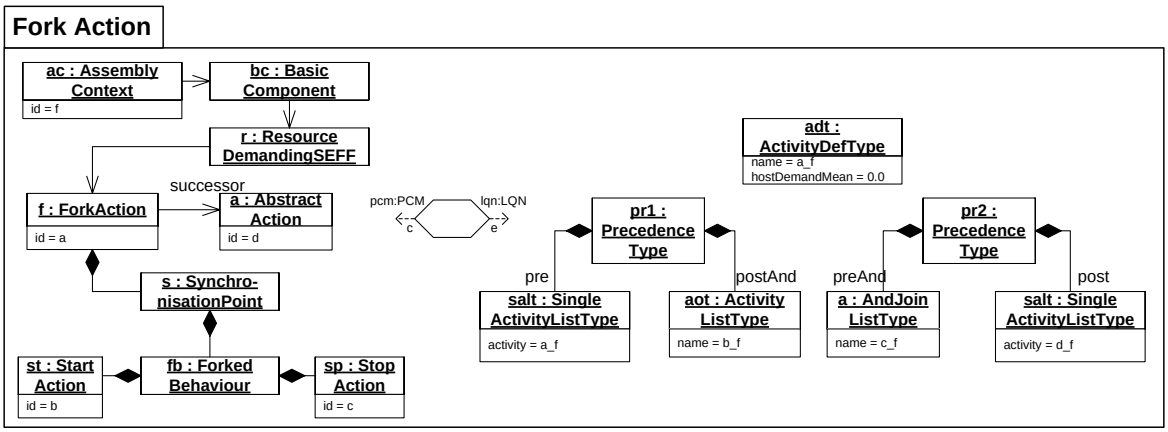


Figure 6.23: Mapping PCM2LQN: `ForkAction`

the possession of passive resources. PCM2LQN maps each `PassiveResource` of a `BasicComponent` to a new task (Fig. 6.24). It has the scheduling discipline `semaphore` supported by LQNs as described in the previous subsection. This task includes two `EntryTypes` `signal` and `wait` to allow acquisition and release of the underlying passive resource. PCM2LQN maps the `capacity` of the PCM `PassiveResource` to the `multiplicity` attribute of the created task. As the PCM `capacity` c is a `RandomVariable`, PCM2LQN uses the expected value $E(c)$ in the LQN.

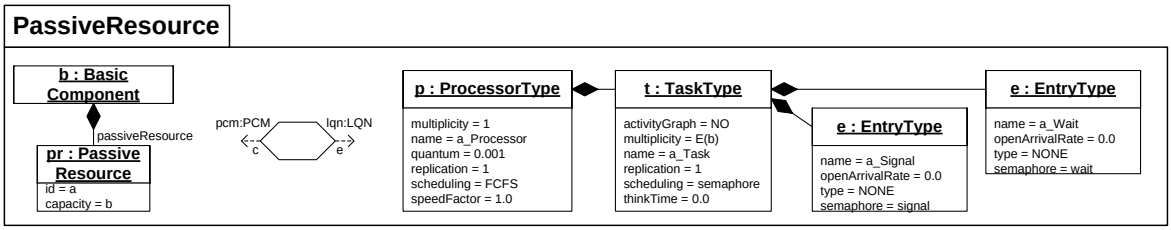


Figure 6.24: Mapping PCM2LQN: `PassiveResource`

For the `AcquireActions` and `ReleaseActions` in RDSEFFs, PCM2LQN creates `ActivityMakingCallTypes`, which issue requests for the task formerly created for the `PassiveResource` (Fig. 6.25 and 6.26). The activities created for `AcquireActions` call the `wait`-Entry, while the activities created for the `ReleaseActions` call the `signal`-Entry. Again the activities are connected to the activity created for the successor actions with a `PrecedenceType`.

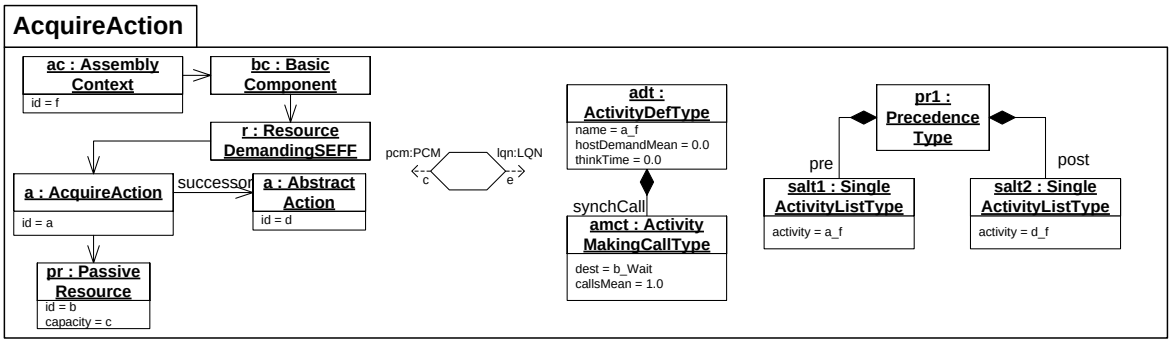


Figure 6.25: Mapping PCM2LQN: `AcquireAction`

Technical Realisation PCM2LQN creates instances of an LQN meta-model in Ecore. It has been generated with EMF from the LQN-XML schema provided with

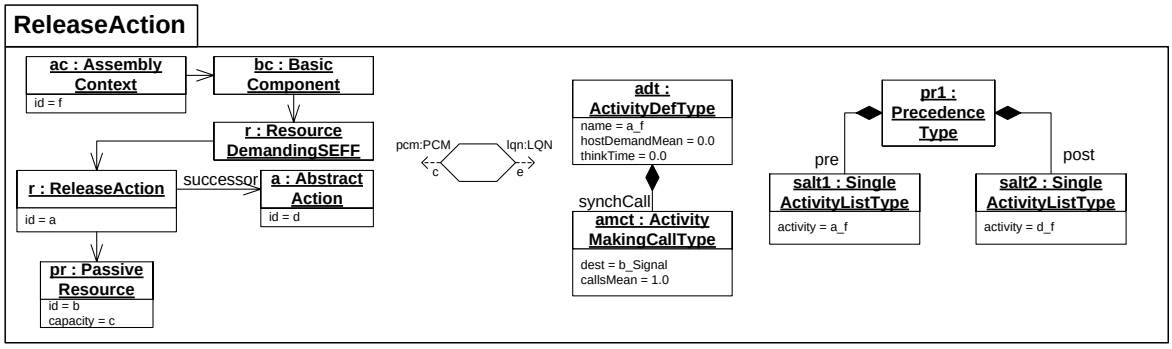


Figure 6.26: Mapping PCM2LQN: `ReleaseAction`

the LQN tools (Version 3.12, cf. [Rea]). PCM2LQN uses an PCM instance with computed contexts as input and traverses it using three EMF-visitors for the PCM usage model, RDSEFFs, and the resource environment. During traversal, it creates instances of the LQN meta-model classes. The visitors use the `ContextWrapper` described in Chapter 6.2.4 to access the computed contexts and to navigate between the different submodels.

Once the visitors have traversed the whole PCM instance, an object representation of the LQN instance has been created. Using the XML serialisation built in EMF, PCM2LQN then saves this representation to an XML file. As the solvers do not support the XML format as input, the tool `lqn2xml` of the LQN tools can be used to convert the XML file back to an older file format based on an EBNF grammar that the solvers accept. Running the solvers then generates the textual output described in Chapter 6.4.4. A graphical visualisation of the results as in the Palladio tools would require parsing the solver output, which is considered future work.

6.4.6 Comparison PCM/LQN

Comparing the PCM with the LQN model points out the benefits and deficits of each approach and yields pointers for future work. The comparison in this chapter is subdivided to features of the meta-models (Tab. 6.1) and features of the existing solvers (Tab. 6.2).

Notice that the comparison of the meta-models uses the PCM with computed contexts instead of the parameterised PCM. The parameterisation of the PCM for different usage profiles and the subdivision of the model for different developer roles can not or only partially be mapped to LQNs. Such a comparison between the parameterised PCM and LQNs would be unfair, as both models aim at differ-

ent goals. The PCM specifically aims at component-based systems and reusable specifications, whereas LQNs aims at arbitrary distributed systems with a focus on behaviours efficiently analysable by QN solvers. Therefore, only the performance modelling concepts such as resources, software behaviour, and communication are compared in Tab. 6.1.

Feature	PCM with Comp. Ctx. (Ecore)	LQN (XSD)
Resource Demand Distribution	General	Exponential
Loop Iteration Distribution	General	Geometrical
Number of External Calls	Deterministic/Stochastic	Deterministic/Stochastic
Stochastic Dependencies	Limited: CollIterAction	No
Workloads	Open/Closed/Multi	Open/Closed/Multi
Behaviour	Seq/Alt/Loop/Fork	Seq/Alt/Loop/Fork
Active Components	No	Yes
Request Queue for Components	No	Yes
Thread Pools	Limited	Yes
Replication of Components	Yes (n AllCtx)	Yes
Communication	Synch/(Asynch)	Synch/Asynch
Forwarding	No	Yes
Phases	No	Yes
Service Centers	Single Server	Multi Server
Replication of Servers	No	Yes
Number of Resource Services	1 per Resource	n per Resource
Scheduling Disciplines	PS, IS, FCFS	PS, PP, HOL, FCFS, RAND
Recursive Service Calls	No	No
Dynamic Architecture	No	No
Exceptions/Failure Behaviour	No	No
Concrete Syntax	Proprietary (UML-like)	Proprietary (Acyclic Graph)

Table 6.1: Comparison PCM/LQN

The PCM supports general distribution functions for resource demands and loop iterations numbers, whereas LQNs only support exponential and geometrical distributions respectively. This might be inaccurate for many distributed systems. However, because of this assumption, LQNs can be quickly solved using analytical methods. As evidenced by the case studies referenced in Chapter 6.4.2, the mean values of the resulting exponential distributions are sufficient to make a rough performance prediction.

With `CollectionIteratorActions`, the PCM includes a limited concept for reflecting stochastic dependencies. However, this is so far only supported by the SimuCom solver, not by the SRE-Solver. LQNs do not support stochastic dependencies and assume independent random numbers.

Workload and behaviour specification are similar in the PCM and in LQNs, but LQNs additionally support active components, request queues with different scheduling disciplines in front of software entities, and have a more convenient

support for modelling thread pools. Active components initiate requests to other components or resources themselves without being invoked by users or other components. This cannot be specified in the PCM. PCM components do not have requests queues, as there queueing only happens at processing resources. Modelling thread pools is easy in LQNs by simply modifying the `multiplicity` attribute of tasks. While it is possible to model thread pools in the PCM, the necessary acquire and release of threads needs to be modelled explicitly in each RDSEFF included in a components, whereas in LQNs this is done automatically.

The support for concurrent control flow is more advanced in LQNs than in the PCM. Asynchronous behaviour is supported by the PCM only indirectly using asynchronous `ForkActions` inside RDSEFFs. All `ExternalCallAction` in the PCM are synchronous calls. LQNs support concepts like forwarding and phases, which have no counterparts in the PCM. Furthermore, LQN processors support replication (i.e., multiple service centers each having an own queue) and multi-servers (i.e., multiple service centers having a single queue), whereas the PCM so far only supports single service centers with a single queue.

LQN processors support multiple services. For example, a processor modelling a hard disk may provide two services ”read” and ”write” with different performance characteristics, which would be expressed in the LQN using two different tasks. This is not possible in the PCM as each resource can only provide a single service. Finally, LQN resources support more scheduling disciplines than the PCM. However, these scheduling disciplines are simplified compared to real schedulers found in todays operating systems and are chosen because they are specifically supported by the queueing network solvers. For the PCM an extension to realistic Windows and Linux schedulers is planned [Hap08].

Some features are neither supported by the PCM nor by LQNs. This includes recursive calls to component services, dynamic architectures with changing component wiring or changing resource environments, and support for failure behaviour, which may have an impact on the perceived performance. These features are pointers for future work.

The concrete syntax of both meta-models as supported by the graphical model editors is proprietary. The PCM uses a notation similar to UML component diagrams and annotated activity graphs, whereas LQNs use an acyclic graph for the tasks and processors, and control flow graphs inside tasks for activities. For the future, a textual concrete syntax for PCM RDSEFFs is planned, which for example shall simplify specifying parametric dependencies.

For both the PCM and LQNs, an analytical solver and a simulation solver are available (Tab. 6.2). However, the analytical solver for PCM instances (SRE) does not support concurrent behaviour and is restricted to providing response times for single user cases as general distribution functions. SimuCom, the simulation solver for PCM instances, supports concurrent behaviour. It can handle multi-class workloads and $G/G/1$ queues. So far it is restricted to PS, IS, and FCFS scheduling disciplines. SimuCom's output includes general distribution functions for response time, throughput, and queue lengths, as well as mean values for utilisation. The distribution functions can be visualised graphically.

Feature	SRE [KBH07]	SimuCom [BKR07]	LQNS [Fra99]	LQSIM [WNPM95]
Workload	-	Open/Closed/Multi	Open/Closed/Multi	Open/Closed/Multi
Queues	-	G/G/1	M/M/n	M/M/n
Scheduling Disc.	-	PS, IS, FCFS	PS, PP, HOL, FCFS, RAND	RR, PP, HOL, FCFS, RAND
Response Time	Gen. Dist.	Gen. Dist.	Mean + Variance	Mean + Variance
Throughput	-	Gen. Dist.	Mean	Mean
Utilisation	-	Mean	Mean	Mean
Queue Length	-	Gen. Dist.	-	-
Result Feedback into Source Model	-	Yes (Annotation Model)	Yes (XML)	Yes (XML)

Table 6.2: Comparison PCM/LQN Solvers

The LQN solvers support mostly similar features except for the $G/G/1$ queues. Both support analysing multi-class workloads. They can handle $M/M/n$ queues with up to five different scheduling disciplines. LQSIM does not support processor sharing, but includes round-robin scheduling instead, for which it needs an additional time quantum specification not needed for LQNS. Both solvers only support predicting mean values plus variances for the different performance metrics. A special feature of the solvers is the result feedback into the XML source model. This for example includes annotating throughput directly to activities inside an LQN.

6.5 Summary

This chapter presented several model transformations from model of the software modelling domain to models of the performance domains. To resolve the parameter dependencies in a PCM, the Dependency Solver performs a transformation on a parameterised PCM instance given a usage model from the domain expert. It propagates inputs specified in the usage model through the system and creates a set of so-called computed contexts for each component, which store information such as

branch probabilities, resource demands, and loop iteration numbers. Running the Dependency Solver is a preliminary step before running other model transformations.

The transformation to SREs allowed mapping PCM instances to a semi-Markov model, which enables fast analysis of single-user scenarios. A specific benefit of the model are the general distribution functions it produces as output for predicted execution times. The transformation to LQNs mapped PCM instances to an extended queueing network model suitable for single- and multi-user scenarios. The LQN solvers can quickly predict the performance of an architecture, but provide only less expressive mean values instead of distribution functions as results. The latter transformation also allowed comparing the PCM to LQN. While LQNs are better suited for asynchronous communication than the PCM, their main drawback is the restriction to exponentially distributed timing values. Both models do not support dynamic architectures, failure behaviour, and recursive component calls. The following chapter will use all transformations and performance solvers in a case study.

Chapter 7

Experimental Evaluation

This chapter describes the experimental evaluation of the modelling languages and performance analysis methods introduced in Chapter 4 and 6. First, Chapter 7.1 explains different types of empirical validations. Chapter 7.2 lists the empirical validations conducted in the context of this thesis. Chapter 7.3 then describes a case study applying the modelling languages and transformations developed for this thesis on a distributed system in detail. The validation compares predictions based on the models created with the new modelling languages with measurements taken from an implementation of the modelled system to assess the prediction accuracy. Finally, Chapter 7.4 reports the results of a controlled experiment with 19 computer science students, who used the developed modelling languages and tools to conduct performance predictions.

7.1 Types of Validations

It is not possible to formally prove that predictions with the models will accurately reflect the performance properties of the real system. First, designers create the models as abstract representations of the system, so that they are manageable and mathematically tractable. The introduced abstraction distorts the accuracy of the predictions. Second, programmers add information and detail to the system when using the models to create an executable implementation, which might alter the system's performance properties. Therefore, this chapter only demonstrates the principle possibility to make accurate prediction based on the new modelling languages, if the available input data for annotating the models is accurate enough and the implementation process does not change the modelled performance significantly.

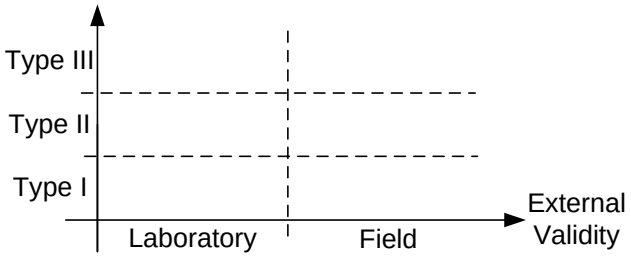


Figure 7.1: Evaluating Model-Based Prediction Methods

There are different forms of evaluations (cf. Fig. 7.1) for model-based prediction methods:

- **Type I (Feasibility):** This is the most simple form of evaluation, where the authors of a method conduct the predictions supported by a tool. It requires an implementation of the model language and the analysis tools. A Type-I study involves comparing predictions with the models with measurements from an implementation of the system. The performance annotations used in the model may either be derived based on estimations or measurements, whereas measurements usually increase the prediction accuracy. This study ensures that the analysis or simulation method for the prediction delivers accurate results under the assumption that its inputs were accurate (examples in [LFG05, Kou06, KBH07]).

- **Type II (Practicability):** Type-II studies evaluate the practicability of a method, when it is used by the targeted developers instead of the method's authors. This includes investigating the maturity of the accompanying software tools and the interpretability of the results delivered by the tools. Type-II studies can also check, whether third parties are able to estimate performance annotations with sufficient accuracy. Such a study may involve students or practitioners. It is desirable to let multiple users apply the method to reduce the influence of the individual and gain general results (examples in [BMMI04, KF05, Mar05]).

- **Type III (Cost-Benefit):** This form of evaluation is a cost-benefit analysis. Applying a prediction method leads to higher up front costs for modelling and analysis during early stages of software development. Model-based prediction methods claim that these higher costs are compensated by the reduced need for revising the architecture after implementation or fixing the code. A Type-III

study checks this claim by conducting the same software project at least twice, once without applying the prediction method and accepting the costs for late life-cycle revisions, and once with applying the prediction method thereby investing higher up front costs. Comparing the respective costs for both projects enables assessing the benefit of a prediction method in terms of business value. Due to the high costs for such a study, it is very seldom conduced (preliminary example in [WS03]).

The external validity of an experimental evaluation refers to the generalisability of the findings to different situations. The results for a study with a low external validity do not hold if the method is applied under slightly different conditions. One main influence factor for the external validity is the system under study. For a high external validity, the analysis of one or more realistic systems instead of specially created example systems is preferable. If the authors of a method create their own example systems under laboratory conditions, they choose specific systems, which emphasise the benefits of their method.

However, field experiments with industry systems are expensive and time consuming. In model-based performance prediction, many researchers analyse their own example systems, which they design so that they represent realistic systems. Whether these systems are indeed representative for a realistic setting needs to be judged by other researchers or practitioners.

7.2 Validations for Parametric Dependencies

Several Type-I evaluations and one Type-II evaluation involving different laboratory example systems have been conducted in the context of this thesis. A Type-III evaluation of the prediction method as well as an application on an industrial-sized system is subject to future work. The different conducted Type-I studies focus on specific parts of the modelling languages and analysis methods contributed in this thesis:

- **Web Server:** In [KF06], a prototypical web server implemented in C-Sharp served as an evaluation example for the predicting the performance using SREs (cf. Chapter 6.3). The study analysed sequences, probabilistic control flow branches, and, in particular, loops involving arbitrarily distributed time consumptions. The new loop concept involving arbitrary distributed number

of iterations was successfully evaluated against common geometrically distributed number of iterations. The web server had been developed in the Palladio group as a student project and had already been used for experimental evaluations in [Koz04, Hap04, Mar05].

- **Web Audio Store:** The study in [KHB06] featured the so-called WebAudioStore as an evaluation example for the newly introduced parameterisation of RDSEFFs. This system was modelled after typical online music stores and included a usage scenario that allowed users to upload MP3 files via the Internet to a database server. It was modelled using UML and an extended version of the SPT profile [Obj05b] and implemented in C-Sharp. Furthermore, an SRE model of the system was implemented manually. The evaluation compared the predicted response time distribution for the upload use case with measurements from the implementation and attested a low deviation. The study in [BKR07] later used the same system, but modelled it with the PCM and performed a simulation of the model instead of SRE analysis. The results were comparable to those in [KHB06].

- **Client/Server App:** In [HKR06], the SRE model was extended to allow concurrent control flow. This allowed modelling thread invocations in single user cases. For evaluation, a simple client/server system was set up, which included a server component offering several functions to a client component which invoked them concurrently. This restricted, abstract setting allowed more control of the system under analysis. The functions implemented different algorithms, such as quicksort, Fast Fourier Transformation (FFT), prime number generation, and Mandelbrot-set calculations. While the model was capable to make predictions in cases with limited memory access, the predictions failed when the algorithms made many memory accesses.

- **Media Store:** In [KBHR08], the MediaStore system was used for evaluating input and output parameter characterisations as well as component parameters (Chapter 4.1.3). It is an extension of the WebAudioStore system with more components and a use case for downloading a set of files from a database. Because it features many concepts introduced in this thesis, the evaluation involving this system will be presented in the following subsections in detail.

While all of the analysed systems are small compared to industry-size applications, they are modelled after typical distributed systems, which are in the target

domain of the PCM. They include performance annotations, which could in principle be similar in larger systems. Evaluations involving industrial size applications are however beyond the scope of this thesis and must be considered future work.

7.3 Type-I-Validation

7.3.1 Setting: Questions, Metrics, Assumptions

The goal of the empirical study described in the following is to assess the benefits of parameterisation concepts introduced in this thesis to the PCM. Therefore, it shall answer the following questions:

- Q1: What is the prediction accuracy of the method?
- Q2: Can the method successfully support the decision for a design alternative?
- Q3: Are the results of different performance solvers comparable?
- Q4: How sensitive is an architectural model for changing parameter values?

The *prediction accuracy* refers to the deviation between predicted performance metrics using models, and measured performance metrics from an implementation. As the PCM and its solvers support distribution functions for performance metrics, there is no straightforward way to assess the deviation as it would be with mean values. Therefore, this evaluation uses statistical goodness-of-fit tests to compare the distribution functions of predictions and measurements. Additionally, point estimators complete the comparison. The hypothesis for Q1 is that a Kolmogorov-Smirnov test [MJ51] will attest no difference between predicted and measured performance metric distributions and that the deviation of the point estimators is below 30 percent as in other performance prediction approaches.

The PCM specifically targets the support of design decisions of a software architect by quantifying the performance properties for different proposed *design alternatives*. The following study does not include multiple design alternatives, but compares using the same architecture with different usage profiles, as the newly introduced parameterisation enables this. The architecture is analysed in two different settings and a service level agreement (SLA) will be evaluated for both settings. The different usage profiles are, however, comparable to different design alternatives. The hypothesis for Q2 is that the method will successfully predict whether the system will violate the defined SLA under the given usage profiles.

As introduced in Chapter 6, there are several *solvers* available for PCM instances, which have individual benefits and deficits. The SRE model and SimuCom aim at producing distribution functions as performance indices, whereas the analytical LQN solver and the LQN simulation provide mean values. However, all solvers should predict the similar mean values, which should be ensured by a correct implementation of the corresponding model transformations. This fact is analysed with Q3. The hypothesis is that the mean values response time predictions from the different solvers do not differ by more than ten percent.

PCM models offer many parameters, which the software architect or performance analyst can adapt during model-based predictions to analyse different trade-offs. For example, the software architect could increase the number of concurrent users or try to decrease the size of data packets transferred over a network. The model under analysis may react differently upon changing different parameters, as the overall performance metrics may be more sensitive to certain parameters. The hypothesis for the according question Q4 is that the MediaStore system is most sensitive to changing the number of users and that there are only linear relationships between changing individual parameter values and the overall performance metrics in this example.

There are several assumptions underlying the following case study, which shall be stated in advance to help the reader assess the validity of the results:

- **Resource demands are timing values:** While RDSEFFs allow the specification of platform-independent resource demand functions for component operations, this feature has not been used in the MediaStore example. With the processing rate of the corresponding processing resource from the PCM resource environment, such platform-independent resource demands can be transformed into platform-dependent timing values. However, in the following timing values are used directly as resource demands and the processing rate of each resource is 1.0.

- **Performance annotations are derived using measurements:** It is assumed that all components of the architecture are already implemented and thus performance annotations such as resource demands can be determined via measuring these implementations. In reality, software architects may combine these measurement-based models with purely estimation-based models for new components, which have not already been implemented. The estimations included in that case might lower the prediction accuracy. However,

the following case study shall not evaluate the developers' estimation skills, therefore measurements are assumed.

- **Linear regression is sufficient for parametric dependencies:** The MediaStore system includes several resource demands, which are parameterised over input parameters. For example, the resource demand for transferring a file over a network link depends on its byte size. It is assumed here that all parameter dependencies are based on linear relationships, which can be determined via several measurements and linear regression analysis. In reality, these relationships might not be linear in all cases.

- **No explicit middleware overhead in the model:** As all performance annotations for the model are obtained by measuring the execution times of individual components on the target middleware and target resource environment, their is no need for explicit model of middleware performance as this is already included implicitly within the measured execution times. This is only a restriction for simplicity of the setting analysed here. Becker's thesis [Bec08] explicitly deals with the influence of middleware features when using PCM models.

- **The usage model accurately reflects realistic workload:** It is assumed that the workload and parameter values of the users modelled in the PCM instance are a realistic representative of the actual workload when running the implemented system. Therefore, for the measurements, the workload modelled in the PCM instance was reproduced using load drivers. In reality, the modelled workload might differ from the workload in reality.

7.3.2 MediaStore

The MediaStore system is a web-based online shop for different kinds of media files, such as audio or video files. The functionality is similar to Apple's iTunes store [App], where users can listen to, download, and purchase a vast variety of music files. Fig. 7.2 shows a combined static and deployment view of the MediaStore architecture.

The architecture consists of a client tier, an application server tier, and a database tier. The client tier represents the users' computers, which access the store via the Internet. The application server is the open-source variant of Sun's Glassfish application server (used because of full EJB3 compliance) and hosts a number of compo-

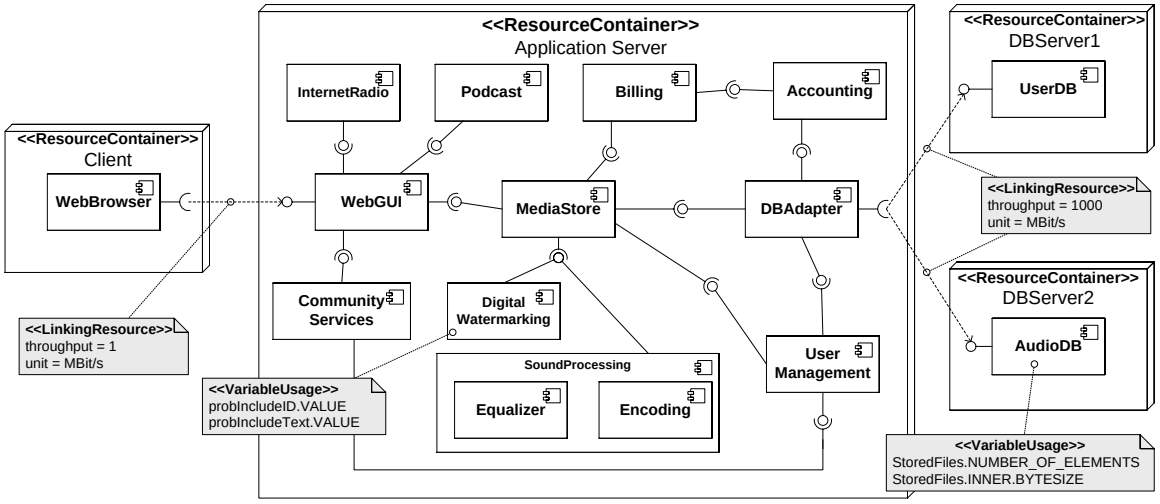


Figure 7.2: Media Store, Static View and Deployment

nents implementing the store's business logic. There are two MySQL databases connected to the application server via Gigabit Ethernet (1000 MBit/s). One database (`UserDB`) stores all customer data, while the other database (`AudioDB`) stores the media files.

Fig. 7.3 shows a dynamic view of the MediaStore architecture, in particular use case 1, where a user downloads a set of files from the store. Via the web browser, the user provides the `WebGUI` component with a query that results in a set of media files from the database. After forwarding the query to the `MediaStore` component, the store searches the `AudioDB` component for the requested files. The matching files are then transferred back to the `MediaStore` component.

For copy protection, the `MediaStore` adds a digital watermark to each file. With the watermark, which is unrecognisable by the user, the store can for example add the current user's ID to the files. If the respecting files would appear elsewhere on the Internet, for example in file sharing services, the user who downloaded the files from the store, could be tracked down with the water mark. The component `DigitalWatermarking` carries out the actual watermarking of the media files. It can be configured to include additional texts, such as lyrics or subtitles, into the files as digital watermarks. This component processes single files, therefore the `MediaStore` component calls it for each file that was requested by the user in a loop (Fig. 7.3). After completing the watermarking, the system sends all files to the user.

Fig. 7.4 depicts the RDSEFFs of the components `MediaStore`, `DigitalWatermarking`, and `AudioDB`. In this case, each component provides only a single

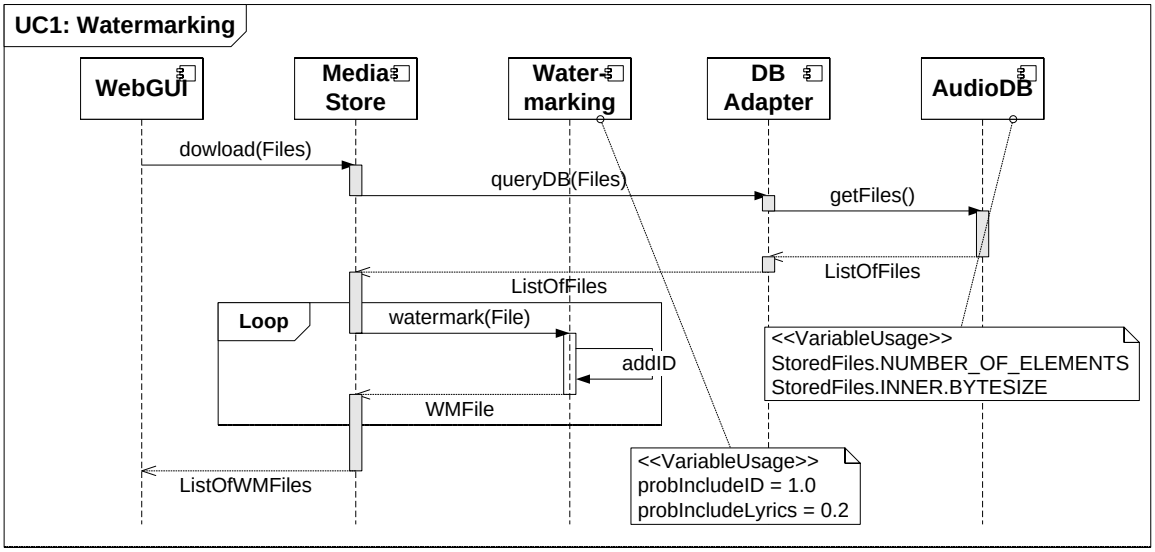


Figure 7.3: Media Store, Sequence Diagram Use Case 1

service and thus has a single RDSEFF. There are several parametric dependencies in these specifications, as described in the following.

The service `download` of the component `MediaStore` forwards the user's the file request to the database via an external service call. Its input parameter characterisation (in this case the number of requested files) depends on the user request (`desiredFiles.NUMBER_OF_ELEMENTS`). The watermarking component is called from this service for each requested file in a loop with a `CollectionIterator-Action`. Thus, the number of loop iterations is the number of requested files. The input parameter characterisation for the external call `watermark` depends on the sizes of the files returned formerly from the external call `queryDB`. A `Set-VariableAction` sets the return value of this service (i.e., the set of files requested by the user), which depends on the file sizes returned by the watermarking component.

The service `getFiles` of the component `AudioDB` realises searching the database for the set of requested files and transferring these files from the hard disk to main memory. Thus, its RDSEFF contains two `InternalActions` with parametric resource demands. The execution time for searching the database depends on the number of files stored in it. Here, it has been approximated with a simple linear functions, which was determined by measuring the time for searching in the MySQL database using different numbers of stored files. Fig. 7.5 shows these measurements and the according linear regression. The same technique has been

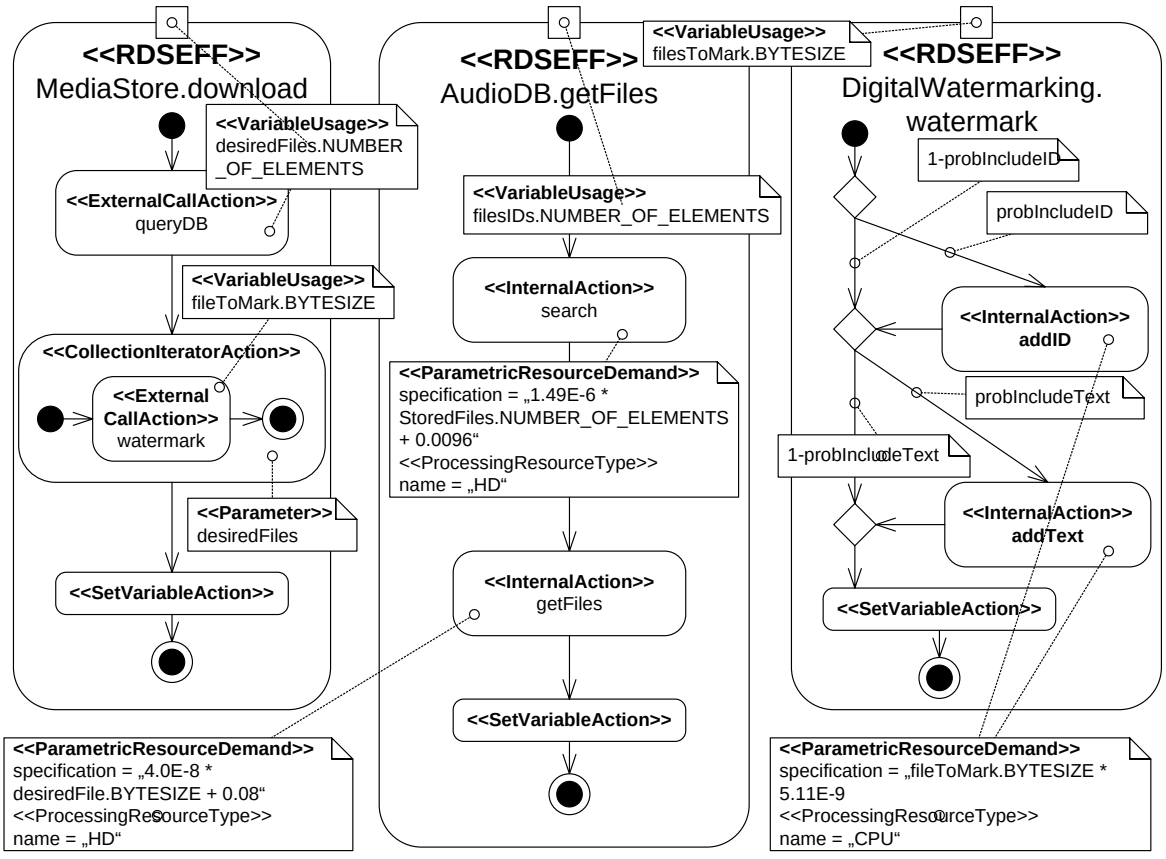


Figure 7.4: Media Store, RDSEFFs

applied to determine the parametric resource demand for the second `InternalAction`, which reads the files from hard disk.

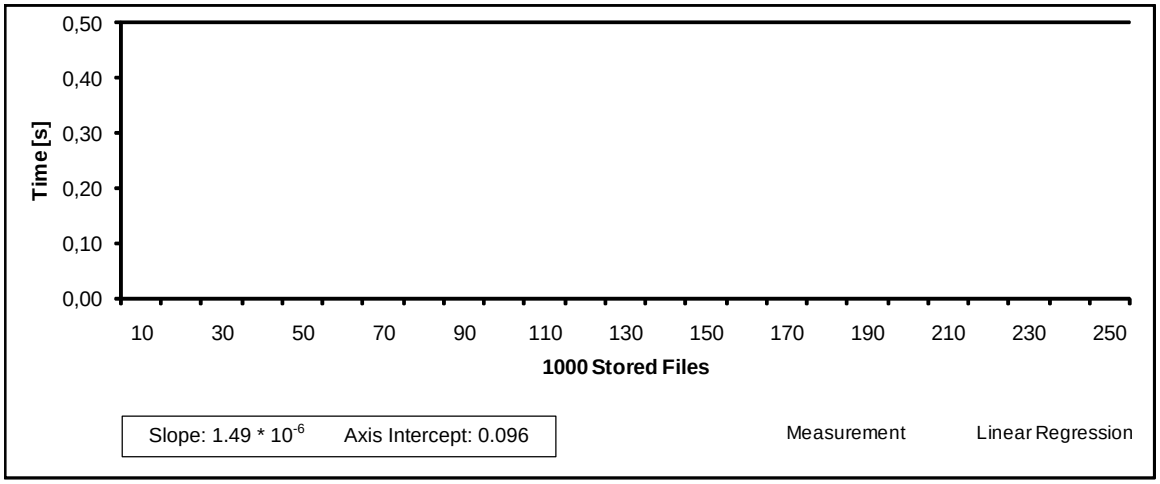


Figure 7.5: Media Store, Linear Regression

Also the service `watermark` of the component `DigitalWatermark` includes a parametric resource demand, because the execution time for the watermarking algorithm depends on the size of the files processed. The linear equation was again determined by measuring the execution time of the algorithm on the target platform using different file sizes.

Notice, that the content of the database has been modelled as a (static) component parameter (`StoredFiles.NUMBER_OF_ELEMENTS`). This component parameter is exposed by the component developer, so that the domain expert or software architect can provide a value for it depending on the usage of the component. Also the size of the stored files can be specified using a component parameter (`StoredFiles.BYTESIZE`), thus this specification can be reused in different designs for different MediaStores and be adapted to a particular setting.

The performance goal for the modelled use case 1 is that the time between the user issuing the request for the files, and the system starting to send the files to the user is less than 8 seconds. This time involves searching the database, reading the files from hard disk, transferring them from the database server to the application server, and watermarking each file. It does not include the network transfer over the Internet to the user. More precisely, the requirements for the architecture include a service level agreement (SLA) stating that at least 90% of the calls have to return in less than 8 seconds. The following performance prediction will check, whether the

Size (MB)	0.0	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0	11.0	12.0
Probability	0.0060	0.0223	0.0466	0.1038	0.1606	0.2038	0.1882	0.1137	0.0685	0.0293	0.0173	0.0093	0.0300

Table 7.1: File size distribution (`Setting1`)

system will be able to fulfil this SLA.

The parameterisation introduced to the component performance specifications in this thesis allows analysing the architecture for different usage profiles. This includes changing the usage models, i.e., how users interact with the system, and changing the component parameters, i.e., the static data the components operate on. Therefore, the following analysis is carried out for two distinctive settings with different parameters.

In `Setting1`, the components in the MediaStore architecture are used to create a music store. The database is filled with 250.000 MP3 files (i.e., `AudioDB.StoredFiles.NUMBER_OF_ELEMENTS = 250000`). Each MP3 file has a size between 1 and 12 MB (cf. exact distribution in Tab. 7.1). The component `DigitalWatermarking` is configured to add user IDs to each passed file (`probIncludeID = 1.0`). Finally, users request 10-14 files with a uniform distribution from the store, which represents a music album.

In `Setting2`, the components in the MediaStore architecture are used to create a video store. The database only contains 10.000 video files (i.e., `AudioDB.StoredFiles.NUMBER_OF_ELEMENTS = 10000`), as there are fewer movies than songs. Each video files is assumed to be highly compressed and has a size between 95 and 105 MB with a uniform distribution. The component `DigitalWatermarking` is configured to add the user ID and subtitles to the video files (i.e., `probIncludeID = 1.0, probIncludeText = 1.0`). Each user seeks only a single movie per request.

Notice that the parameterisation also allows using individual components of the MediaStore architecture in different systems, instead of only using the same system with different parameterisations. For example, the component `DigitalWatermarking` does not depend on the `MediaStore` component and could also be used in an online book shop. Then, the same RDSEFF can be used for performance predictions because of its parameterisation, which allows adapting the specification to the inputs from the book shop. To restrict the complexity of the evaluation in this section, performance prediction is carried out here only for the same architecture with two different usage profiles.

For the implementation of the MediaStore, the model-to-code transformations developed in [Bec08] have been used to generate code skeletons, which were com-

pleted with the respective business logic (e.g., for watermarking) manually. The implementation is based on EJB3. The model-to-code transformation also generated build scripts, deployment descriptors, configuration files, and a test client. Using AspectJ, measurement probes were weaved into the code. A pre-test run ensured that these probes did not distort the measured execution times significantly. During the actual measurements, where the modelled usage profile was reproduced using the test client, 500 measurements of the response time were recorded to get distribution functions for both settings.

7.3.3 Results

This subsection compares performance measurements of the MediaStore implementation with performance predictions of the MediaStore model produced by the different solvers. It will first describe the results for each individual solver (SRE, SimuCom, LQNS, LQSIM) to answer the formerly defined questions Q1 and Q2, before comparing the predictions to answer question Q3. Question Q4 about the sensitivity of the architecture will be tackled in the next subsection.

Stochastic Regular Expressions For the SRE prediction, the model transformations described in Chapter 6.2 and 6.3 produced a Stochastic Regular Expression consisting of 28 symbols from the MediaStore PCM instance in about 300 ms. Running the SRE-Solver (ca. 50 ms) yielded probability distributions for both settings of the MediaStore.

Fig. 7.6 shows the results for `Setting1` consisting of two histograms on the left hand side and the corresponding cdfs on the right hand side. Each figure includes the distribution for the measurements (straight line) and the predictions (dashed line).

The graphs for measurement and predication widely overlap. However, the predicted data is more smooth than the empirical data due to the approximated distribution functions and the involved convolutions. The response time with the highest probability is 6.0 seconds for the measurements and 5.7 seconds for the predictions. The expected values for both distributions are 6.23 seconds and 5.61 seconds respectively (less than 10 percent deviation).

At a significance level of $\alpha = 0.01$, a KS-test was not able to reject the null hypothesis of prediction and measurement having the same underlying probability distribution. The compliance of the SLA of 90% of calls returning in less than 8 sec-

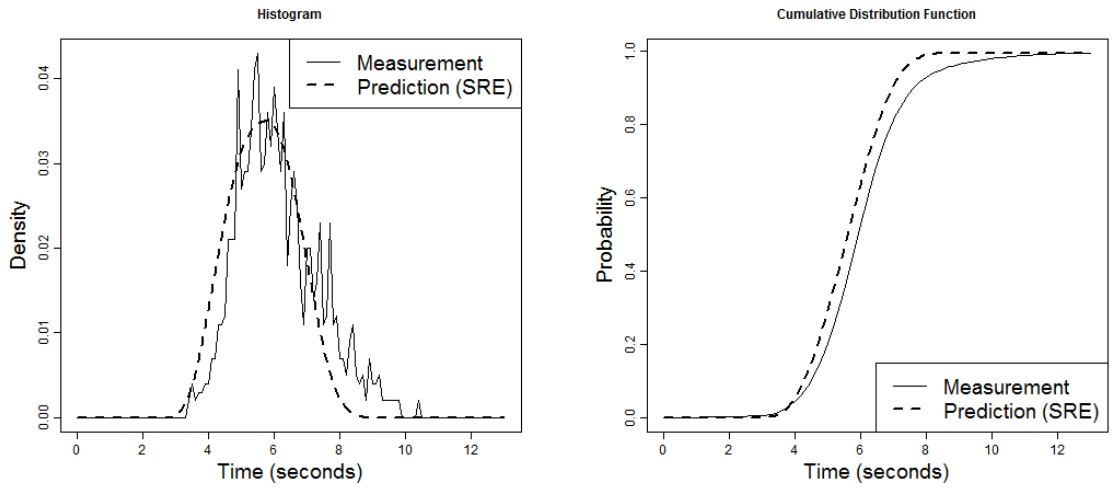


Figure 7.6: Response Time MediaStore Setting 1

onds was correctly predicted, as the SRE-Solver predicted 98% of calls returning in less than 8 seconds and the measurements confirmed this, as actually 92% returned in less than 8 seconds.

Similar graphs illustrate the results for `Setting2` (Fig. 7.7). The predicted values clearly show the uniform distribution of the files sizes used in this setting. The measured values spread more than the predicted values supposedly because of still inaccurate handling of the higher network load in the prediction model. The most probable response time is 9.6 seconds (measured) and 9.4 seconds (predicted), whereas the expected values are 9.74 seconds and 9.58 seconds respectively.

As in `Setting1` a KS-test was not able to show that the underlying probability distributions of measurements and predictions are different. Here, both predictions and measurements reported a violation of the desired SLA of 90% of calls returning in less than 8 seconds. The measurements showed that 90% of the calls return only in less than 11.4 seconds and the predictions yielded 10.4 seconds and thus correctly forecasted the violation of the SLA. Therefore, `Setting2` is not capable of fulfilling the required SLA.

SimuCom The SimuCom solver for PCM instances [Bec08] additionally produced predictions for the MediaStore system, which are presented in the following to assess the accuracy of the different solvers. Running SimCom's model transformations

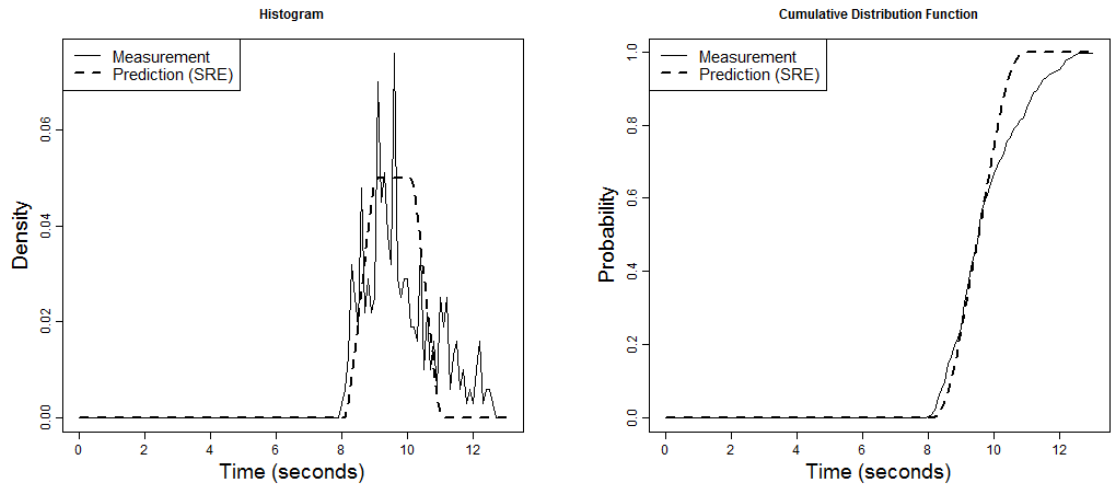


Figure 7.7: Response Time MediaStore Setting 2

and completing a simulation run that produced adequately many data points took ca. 30 seconds for the MediaStore system, which is significantly slower than running the SRE-Solver (below 1 second). In this simple single user scenario, the SRE-Solver is therefore advantageous over the simulation. However, SimuCom becomes advantageous in the often more interesting multi-user scenarios.

Fig. 7.8 for `Setting1` and Fig. 7.9 for `Setting2` appear similar to the figures for the SRE-solver. Measurements and predictions widely overlap. The predicted values are not as smooth as from the SRE-solver, because of the different solution mechanisms. The expected values for probability distributions are 6.17 seconds (`Setting1`) and 9.60 seconds (`Setting2`), whereas the measurements yielded 6.23 seconds and 9.74 seconds respectively. The deviation between predictions and measurements is therefore below 10 percent.

For both settings a KS-test was not able to reject the null-hypothesis of the prediction and measurements having the same underlying distribution functions. As for the SRE-Solver, the compliance and violation of the SLA was correctly predicted with SimuCom.

LQNS/LQSIM Using the PCM2LQN model transformation, an LQN was generated from the MediaStore PCM instance (duration: ca. 200ms). Fig. 7.10 illustrates the generated LQN. There is an LQN task for each usage scenario

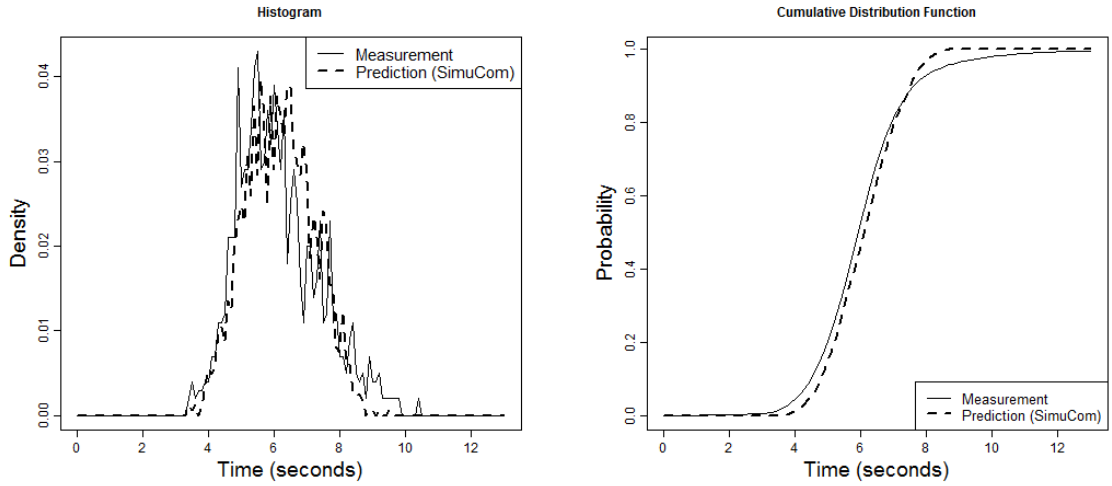


Figure 7.8: Response Time MediaStore Setting 1

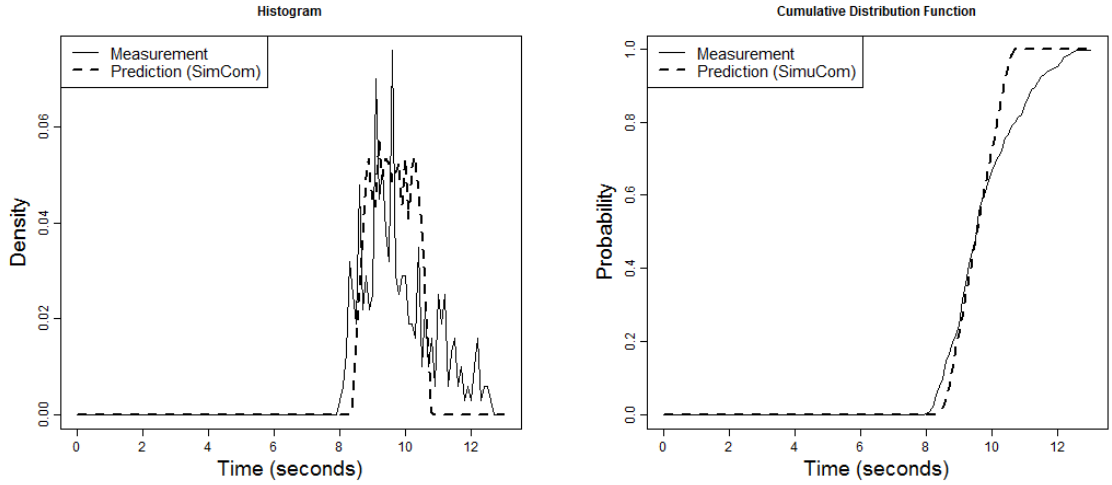


Figure 7.9: Response Time MediaStore Setting 2

(`DownloadFiles`), each task, each loop action, and each each resource. The tasks that do not directly access resources but delegate calls to other tasks, run on dummy processors, where they produce no resource demands. These processors are necessary for the LQN to be syntactically correct. Each task contains at least one entry. Activities and precedence specified by task graphs for these entries are not shown in the figure.

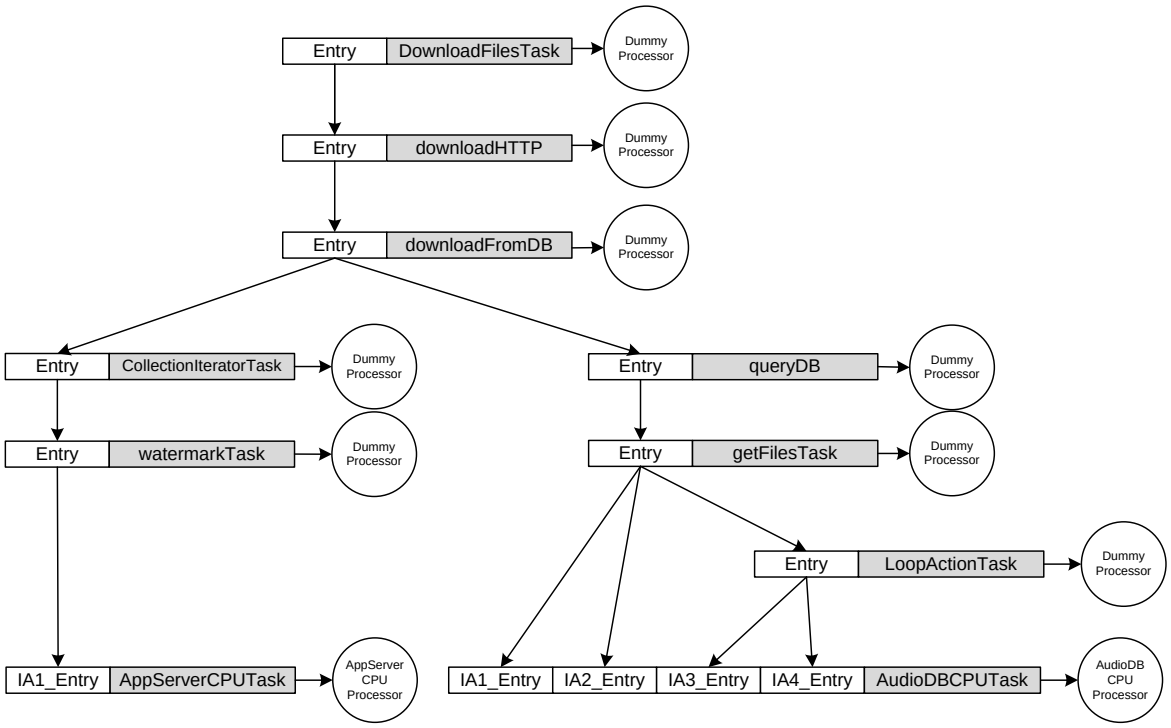


Figure 7.10: Layered Queueing Network for the MediaStore

Running LQNS or LQSIM on the generated LQN took less than 10 ms in both cases and yielded the usual performance metrics as mean values of exponential distribution functions. Therefore, only point estimators of the measurements can be compared with the LQN predictions. LQNS predicted the response time for `Setting1` of the MediaStore as with a mean value of 6.47 seconds (measurement: 6.23 seconds), and for `Setting2` with a mean value of 10.26 seconds (measurement: 9.74 seconds). Thus the deviation in both cases was less than 10 percent. LQSIM predicted the response time for `Setting1` of the MediaStore as 6.37 seconds and for `Setting2` as 10.20 seconds. There are only very small deviations both from the LQNS predictions and the measurements.

The metrics for throughput and resource utilisation are less interesting in the

single-user scenario and have thus been omitted here. It is not possible to answer question Q2 about the compliance to the SLA of 90 percent of the calls returning in less than 8 seconds with the LQN solvers adequately, because they only support mean values for results, but not to determine arbitrary quantiles.

Comparison To assess the different solvers and answer question Q3, Fig. 7.11 shows the predicted mean value response times of each solver as well as the measured mean response time. Deviations between measurements and predictions are limited for all predictions, the error is less than 10 percent in all cases. The SRE predictions deviate the most from the measurements in both cases. This might be a result of the high number of convolutions involved in this case, which introduce rounding errors. The LQNS and LQSIM predictions are higher than the measured values in both cases. This again might result from rounding errors introduced by the transformation when deriving the expected values from the probability distributions in the PCM instance.

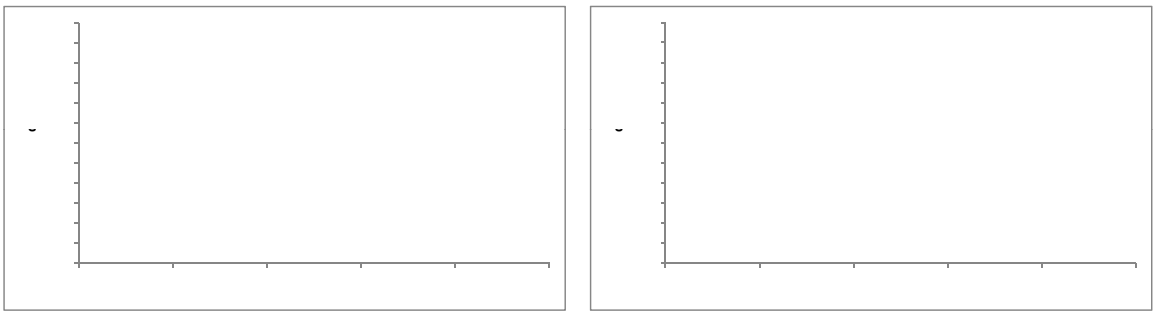


Figure 7.11: Response Time (Mean Values)

Besides the prediction results, the execution time of running each solver can also be compared. The LQN tools (both LQNS and LQSIM) are the fastest solvers available for PCM instances, as they analyse the MediaStore example in less than 10 ms. However, they only produce less expressive mean values. The SRE-Solver predicts the performance in ca. 50 ms, but is restricted to single-user cases. This limitation is not present for the SimuCom simulation, whose execution time is the longest. It includes substantial processing time for the model-to-code transformations, and the simulation time can be adjusted by the user for a desired accuracy. To get the result presented before, running SimuCom took about 30 seconds, where the model transformation consumed about 20 seconds and the simulation about 10 seconds. However, other than the LQN tools, both PCM tools have not been optimised yet.

7.3.4 Sensitivity Analysis

At the beginning of this section question Q4 asked for the sensitivity of an architecture to changes in parameter values of the models. The following presents a sensitivity analysis resulting from more than 50 simulation runs, where single parameters of the MediaStore were adjusted to different values to determine their impact on the overall predictions.

The analysis is deliberately kept simple to allow an easy comprehension of the results. Only a single variable in the PCM instance is changed at a time to determine its sensitivity, while all other variables are kept at the default values described before. This allows determining the value ranges for certain parameters, which do not violate the SLA. However, it does not allow analysing the effect of changing multiple variables at a time, as combinatorial effects are excluded from the analysis. Furthermore, changing variables in a PCM instance could involve changing the form of probability distributions (for example from a uniform to a normal distribution). In the analysis performed here, the form of the probability distributions is not altered, only the expected value of these probability functions is changed by adding or subtracting constants to the distribution.

As the predicted response times from the PCM solvers are given by probability distributions, it would require the sensitivity analysis to compare these probability distributions to each other. To allow better comprehension, the results in the following are compared by point estimators of the probability distribution. The performance goal of the MediaStore scenario was that 90% of the calls return in less than 8 seconds. Thus, the sensitivity analysis compares the 90% quantiles of all predicted response times to determine the parameter range where the SLA would be violated.

Fig. 7.12- 7.14 show the results for changing variables in the usage model, the system model, and the resource environment model respectively. The domain expert would make changes to variables in the usage model. In the MediaStore system, the domain expert can adjust the number of requested files as well as the user population. Fig. 7.12 shows the impact of these changes plotting the expected number of requested files (left hand side) and as well as the user population (right hand side) against the 90%quantile of the predicted response time.

While `Setting1` had a default expected number of requested files of 12, the figure shows that the SLA would be violated if users would request a probability distribution with an expected value of 14 files from the store, while leaving all other variables at their default values. The slope of the curve is 0.35. The MediaStore is

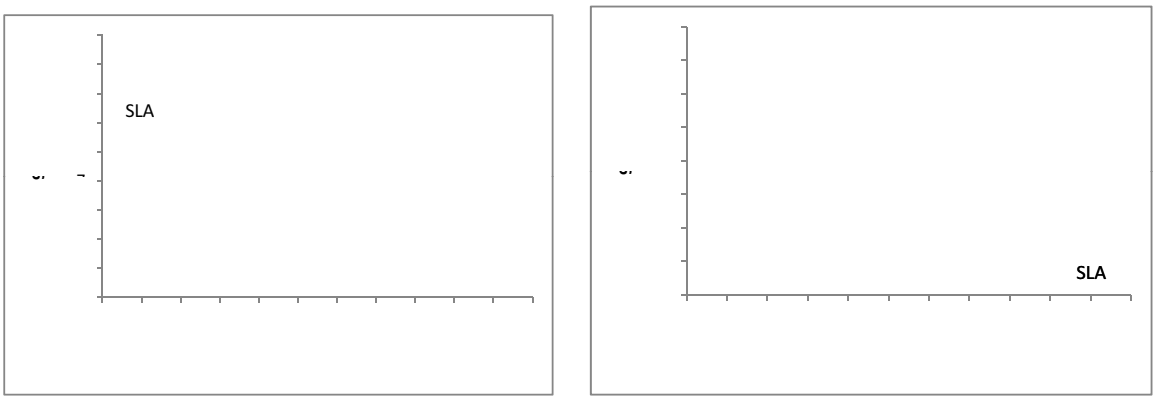


Figure 7.12: Sensitivity Analysis: Media Store Usage Model

however much more sensitive to the number of users concurrently issuing requests, as the right hand figure depicts. A number of two or more concurrent users already violates the SLA, so that the system cannot reply to the requests in less than 8 seconds. Here, the slope of the curve is 7.13, therefore there is a substantially higher sensitivity to the user population that for the number of requested files.

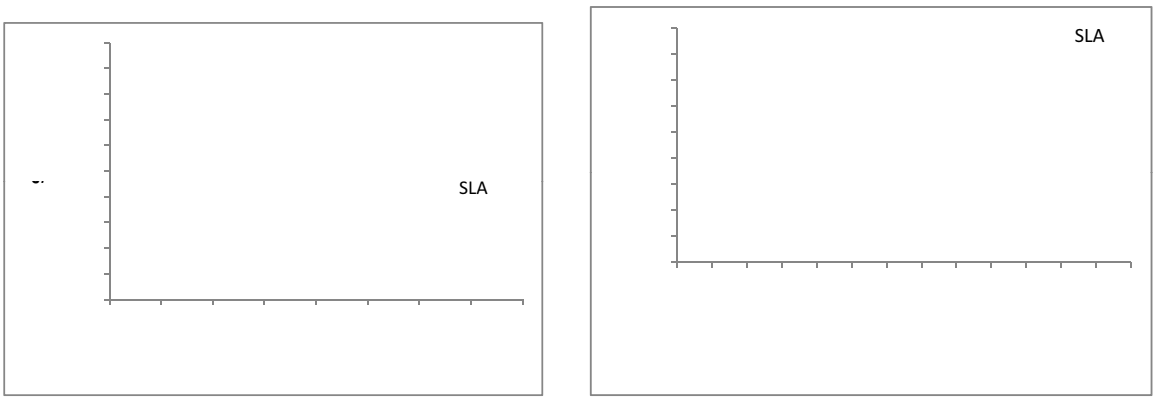


Figure 7.13: Sensitivity Analysis: Media Store System

Fig. 7.13 illustrates the predictions for adjusting the component parameters used in the MediaStore system to different values. Increasing the file size distribution from the default expected value in `Setting1` of 5.5 MB to expected values above 6.0 MB for higher audio or video quality would violate the SLA (Fig. 7.13, left hand side). The slope of this curve is 0.68, therefore higher than the slope for the number of requested files. However, the MediaStore system is not sensitive for the number of files present in the `AudioDB`, which alters the execution time for search requests (Fig. 7.13, right hand side). Increasing the number of files from 250000 to 400000 im-

plies almost no effect on the overall predicted response time of this use case, because this parameter is involved in a resource demands which is too small to significantly change the overall execution time.

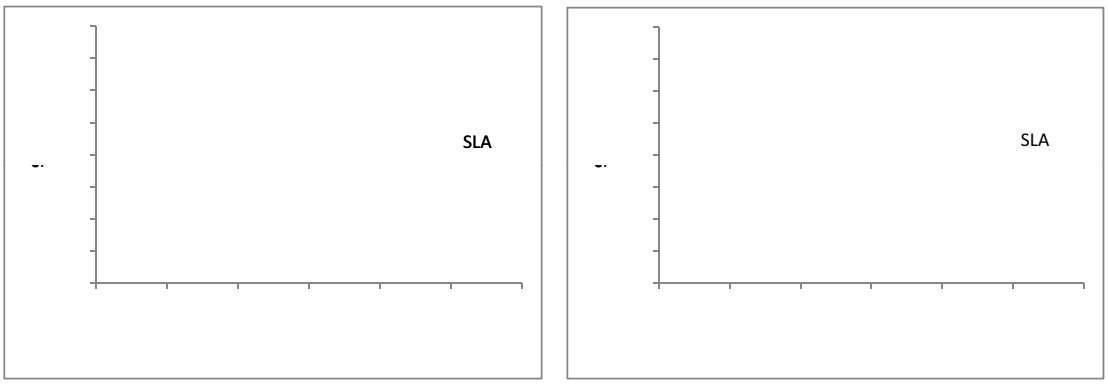


Figure 7.14: Sensitivity Analysis: Media Store Resource Environment

Finally, Fig. 7.14 illustrates the predicted effect of changing the processing rates of the CPUs in the resource environment to assess the inclusion of faster (or slower) hardware. Changing the `AppServer` CPU, which performs watermarking the media files, almost shows no effect on the overall response time, as most processing power is requested from the `AudioDB`. It is even possible to use a CPU with half of the processing rate (0.5) than the default CPU without violating the SLA. Changing the `AudioDB` CPU, which is used when searching for files and retrieving them from hard disk, has a higher impact. With a CPU twice as fast, almost a bisection of the predicted response time can be achieved.

7.3.5 Discussion

The formerly presented results have shown that the MediaStore system is most sensitive to the number of users concurrently accessing the store. To allow concurrent access by multiple users, the store architecture would have to be changed significantly. Some of the variables analysed above cannot be easily changed by the software architect when designing the system. For example, the usage model is often part of the requirements and its parameters can only be changed by renegotiating the requirements. Often, also the resource environment is fixed or hardly changeable without substantial costs.

However, a parameter controllable by the software architect is the size of the files present in the `AudioDB`. An additional component could be introduced to lower the

bitrates of the stored files, thereby decreasing the overhead for reading the files from disk and transferring them over the network. In this case, the performance analysis would have given the software architect a substantial hint on where to change the system.

As already mentioned, the sensitivity analysis presented here has not studied the effect of changing multiple variables at a time, because that would quickly increase the solution space beyond comprehensibility. However, after the single-variable sensitivity analysis a goal-oriented multiple-variable analysis could be performed, for example by increasing the user population and decreasing the file sizes at the same time. This is regarded as future work.

In the future such an analysis should also be carried out with systems with more complex parameter dependencies. The MediaStore features only linear dependencies, which lead to linear effects when adjusting the parameter values. In reality, there are also often non-linear dependencies, which can be expressed with the StoEx language. Assessing their sensitivity is usually more interesting than the for linear dependencies.

7.4 Type-II-Validation

Complementing the Type-I-Validation in Chapter 7.3, the goal of the Type-II-Validation presented in this section is to empirically evaluate the practicability of the PCM and its tools from the user's point of view. This validation has been realised as a students experiment as part of a Master's thesis by Anne Martens (cf. [Mar07]).

The following describes this study with a special focus on results concerning the parameterisation concepts contributed to the PCM in this thesis. Chapter 7.4.1 introduces the setting of the study, defines questions to be answered to achieve the studies goal, and lists assumptions of the study. Chapter 7.4.2 explains the experiment's design and the systems under study. Chapter 7.4.3 describes and discusses the results of the study, before Chapter 7.4.4 concludes the Type-II-Validation by discussing the study's internal and external validity.

7.4.1 Setting: Questions, Metrics, Assumptions

The following questions were asked before the study to direct its conduction (slightly adopted from [Mar07]):

- Q5: What is the accuracy of the method when applied by third parties?

- Q6: Can third parties successfully assess a design alternative with the method?
- Q7: What causes the achieved prediction accuracy by third parties?
- Q8: What is the duration for applying the method by third parties?

Q5 and Q6 are closely related to Q1 and Q2 from the former section. Metrics for Q5 compare prediction made with a model against a sample solution for the experiment's tasks using statistical methods. Metrics for Q6 compare a ranking of design alternatives based on model predictions with a ranking from a sample solution. The hypotheses for both questions are the same as for Q1 and Q2: the deviation between the response time mean values is below 30% and third parties can rank the design alternatives correctly.

Q7 asks for causes of the achieved prediction accuracy. Answers for the question shall be provided by quantitative and qualitative metrics. As the participants had to pass an acceptance tests of their models, which ensured a minimum quality level, quantitative metrics count the number of failed acceptance tests and the number of recorded interpretation problems. Qualitative metrics are collected using questionnaires asking the third parties for their explanation for causes of the achieved prediction accuracy. There are several possible factors influencing the predictions, such as comprehensibility of the meta-model, usability of the supporting tools, and adequacy of the modelling language's concrete syntax. The hypothesis for this question is that main cause for inaccurate predictions are the prototypical tools.

Q8 analyses the effort needed to conduct a performance prediction with the method. While it might be possible to achieve very accurate predictions with a method when spending a large amount of time, this is not practical due to the usually limited resources available for performance modelling and prediction in the software industry. Therefore, metrics for this questions measure the duration for applying the method and break it down to individual tasks, such as modelling, running the simulation, fixing errors, etc. The hypothesis is that a reasonable performance prediction for a small system can be carried out in less than a day.

In this thesis, the focus is on checking the practicability of the newly introduced parameterisation concepts. Therefore, when describing the results of the Type-II-Validation special emphasis is laid on analysing whether the parameterisation caused inaccuracies in the predictions or was responsible for a longer duration of the performance prediction.

Several *assumptions* had to be made to carry out the Type-II-Validation as a student experiment due to organisational constraints. They are listed here to allow the reader judging the validity of the study. A thorough discussion of the experiment's internal and external validity follows after presenting the results of the study. Notice that the assumptions listed in Chapter 7.3 also apply for the Type-II-Validation. Additional assumptions are:

- **No use of the PCM's role concept:** Each participant of the student experiment fulfils the roles of component developer, software architect, system deployer, domain expert, and QoS analyst at the same time. To increase the number of data points the modelling and prediction tasks were not divided among different developer roles. Therefore, the benefits of the PCM's role concept cannot be assessed with this study.

- **No estimations by the participants necessary:** It is assumed that all data necessary to create performance annotations is already available for the experiment, i.e., the participants do not have to estimate values or perform measurements with prototypes. This assumption limits the effort for applying the method and allows a more thorough analysis of the overall modelling and prediction.

- **No comparison between predictions and measurements:** Instead of comparing predictions and measurements, in this study predictions of the experiment's participants were made with predictions of a sample solution for the experiment's tasks. Due to the immaturity of predicting the performance of concurrent control flows with the PCM, this assumptions has been introduced to allow modelling concurrent component interactions in the experiment's tasks. It remains future work to improve the solvers to correctly predict the performance in these cases.

7.4.2 Experiment Design

The Type-II-Validation has been conducted as a controlled experiment with a number of students each applying the method. Having multiple test persons instead of just a single one is a mean to control the distorting influence of the person's individual capabilities.

It was furthermore decided to compare the Palladio prediction method with the established SPE method [Smi02]. This allowed quantifying the proposed improve-

ments of Palladio over an existing method. The SPE method was chosen, because a former study [KF05] had attested it the most maturity from the available performance prediction approaches.

The independent variable of the controlled experiment was the applied method (i.e., Palladio or SPE). The dependent variables for both methods were aligned with the metrics described at the beginning of the section. This included the predicted response times for the systems under study, the ranking of design alternatives, and the duration for all involved tasks as quantitative measures. Furthermore, questionnaires for each participant helped collecting additional qualitative data, for example a subjective assessment of both methods.

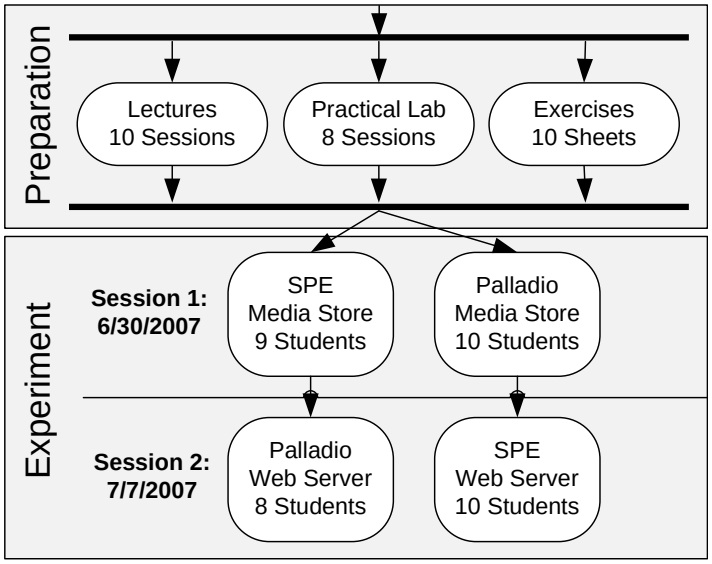


Figure 7.15: Design of the Experiment

Fig. 7.15 depicts the different parts of the experimental design, which includes a preparation phase lasting several weeks and then the conduction of the experiment on two days. The following will detail on these parts.

Participants and Preparation As mentioned before, 19 computer science students participated in the experiment [Mar07]. No practitioners participated in the experiments due organisational constraints (i.e., it requires substantial costs to have practitioners apply the methods). While the students had limited programming experience, most of them had at least participated in lectures about software engineering

and software architecture. They were familiar with the UML, which was useful as the models created during the experiment are similar to UML models.

The study was conducted as part of a university lab course on software performance engineering. The students were motivated to complete the course successfully, because they received grades for it.

Because the students were not familiar with both Palladio and SPE before starting the course, 10 lectures sessions and practical exercises were held to prepare them for the experiment (Fig. 7.15). 2 lectures covered SPE and 5 lectures covered Palladio due to its higher complexity. The students had to hand in solutions for weekly exercise sheets, which were graded to ensure their familiarity with the methods and their tools.

Experiment Plan and Execution The experiment itself was designed as a change-over trial [WRH+00] applying the prediction methods on two different systems (Fig. 7.15). The students were divided into two groups and each student of each group applied one method to one system during each session. In the second session, each student switched the prediction method as well as the analysed system. This allowed tracing back differences in the results of each participant to specifics of the analysed system instead of interpreting differences in the results as specifics of the applied method.

To ensure balanced capabilities of both groups, the participants were assigned to the groups based on their grades for their solutions to the exercise sheets. Students from the better half and the worse half were assigned randomly to each group. This limited the effect of having one more capable group, which would distort the measurement of the dependent variables.

The experiment sessions had a maximum time limit of 4.5 hours to model the systems with the corresponding tools of each method and conduct the performance prediction. The time limit shall reflect the situation in the software industry, where the resources for performance modelling and prediction are usually limited. However, during the execution of the experiment the time limit was loosened to 6 hours, as several participants were not able to hand in appropriate solutions in the allotted time.

To ensure a minimal quality of each participant's solution and to avoid trivial solutions, acceptance tests during the experiment sessions checked the models created by the students. The students could not continue with other experiment tasks until they successfully passed an acceptance test. This ensured more data points

for the dependent variables as the number of useless outliers was decreased. Furthermore, it motivated the students to produce good solutions. The acceptance tests only checked for a coarse grain compliance with the sample solution (both predictions and model structure) of the experiment's tasks, but did not enforce strict compliance. Therefore, the results presented later still exhibit deviations between the student's results and the sample solution.

During the experiment sessions at a university lab, four experimenters supervised the conduction. Participants were allowed to ask questions to the experimenters if they ran into problems that prevented continuing the experiment. The experimenters documented the given advice (cf. [Mar07]), so that their influence on the participants results can be judged independently.

Systems under Study [1 page] The students analysed one of two systems during each experiment session. The MediaStore system is similar to the system analysed in Chapter 7.3, while the WebServer system is a prototypical component-based server for downloading HTML-files with multimedia content. The experiments' task descriptions contained UML component diagrams of the static system architectures, as well as sequence diagrams of performance critical use cases with performance annotations. The participants were asked to model the systems with the tools of each method and conduct performance predictions for different design alternatives and usage profiles as described in the following.

The MediaStore system was slightly adapted from the formerly presented version. It did not contain a database adapter and all components were allocated onto a single server. There were two different performance-critical usage scenarios running in parallel on the server. The first scenario involved downloading a set of files from the system, which required watermarking each file. The second scenario modelled the upload of single media file to the store's database.

Additionally, the participants had to analyse two different usage profiles consisting of different parameter characterisation in the usage model and different component parameter characterisation in the system. The first usage profile modelled a single user executing the usage scenarios alternately. The second usage profile modelled multiple users in an open workload using the system concurrently and increased the number of requested files and the number of downloads.

The experimental tasks included five design alternatives for the MediaStore and required the participants to analyse each design alternative for each usage profile (i.e., a total of 12 predictions including the original system and five alternatives). The

alternatives included i) introducing a cache component, ii) introducing a database connection pool, iii) allocating the database component to another server, iv) compressing the media files, and v) introducing a broker look-up (also see [Bec08]) between the components. The students were required to rank these design alternatives according to their impact on the performance of the system. In the sample solution created by the experimenters before the conduction of the experiment, alternative i) (cache) and iv) (compression) yielded the fastest response times.

The WebServer system allowed users to download static and dynamic HTML files. It consisted of six components allocated to a single server. The system specification contained only a single usage scenario, which was however complexer than the one for the MediaStore because it involved multiple requests and control flow. Furthermore, the experiments' task included five design alternatives, which were comparable to the MediaStore design alternative. They were i) introducing a cache for dynamic content, ii) broker look-up, iii) parallel logging, iv) using an additional second server, and v) introducing a thread pool. The sample solution revealed that introducing the second server (alternative iv) allowed the fastest response times in this scenario.

As for the MediaStore system, the participants had to analyse two different usage profiles. The first profile involved a single user workload, a specific proportion of dynamic and static content, and a specific distribution of the number of dynamic objects and the file sizes. In the second usage profile, multiple users accessed the WebServer in an open workload, requested more static files, and fewer dynamic objects.

Tools For the sample solution both systems were modelled in all design alternatives and with both usage profiles using the tools provided by the methods.

The *PCMBench* (Fig. 7.16) includes a number of graphical editors to create PCM instances. The concrete graphical syntax of these editors is aligned with UML component diagrams, UML activity diagrams, and UML deployment diagrams. One aim of the experiment is to find out whether the use of these editors is intuitive for the participants.

After the user has completed the modelling process, the PCMBenchs validates the model instance by checking several OCL constraints and reports errors or omissions to the user. Running the SimuCom, as performed by the participants of the experiment, yielded a set of experiment data, which the user can visualise for example as histograms or cdfs. The experiment also checks whether the simulation result

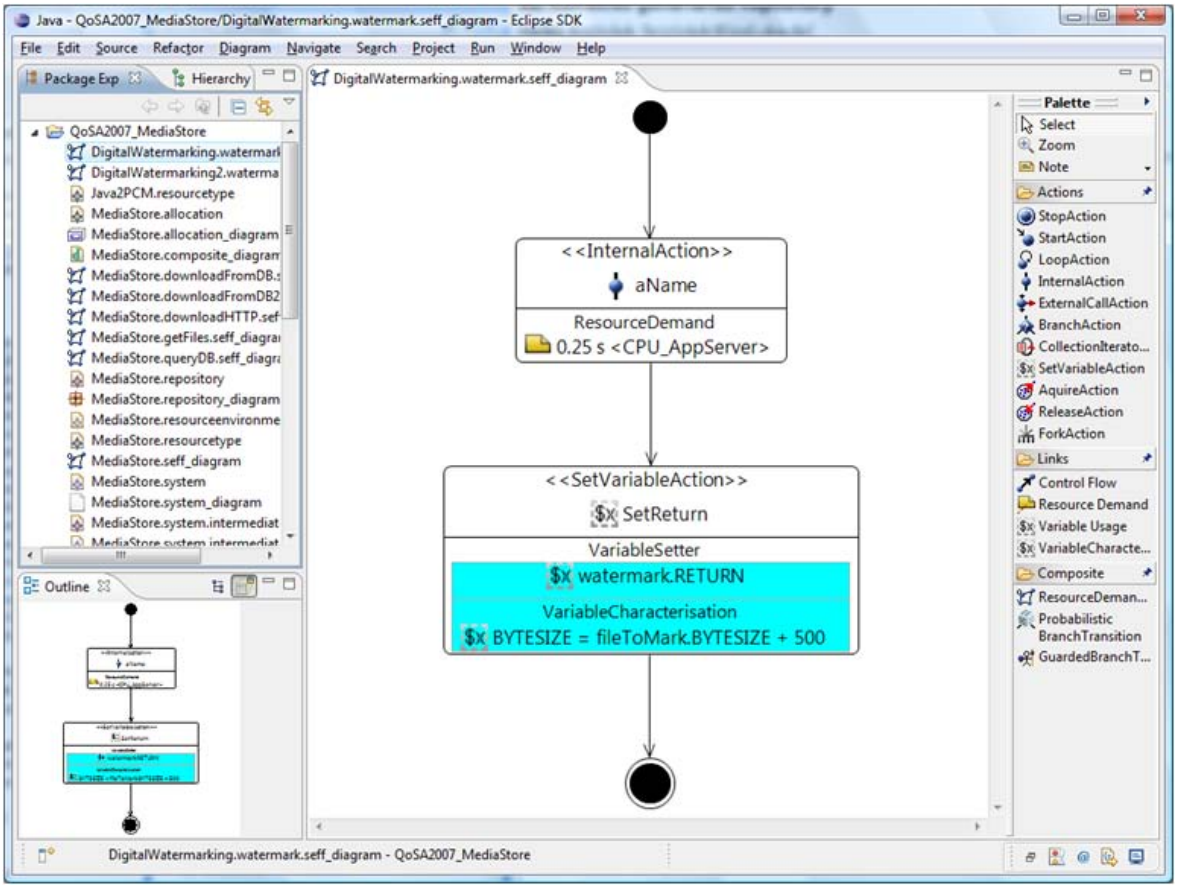


Figure 7.16: Screenshot PCM-Bench

is comprehensible and useful for the users.

The *SPE-ED* tool enables creating and analysing models for the SPE method [Smi02]. The user can create annotated control flow graphs to model a dynamic view of a software architecture. Furthermore a so-called overhead matrix specified the time for executing specific actions (such as CPU access, reading from a database etc.). The concrete graphical syntax is proprietary and not aligned with UML. The user can analyse the created models with QN solvers or a discrete-event simulation.

7.4.3 Results

The following presents the results to answer the questions Q5-Q8 raised in Chapter 7.4.1.

To assess the prediction accuracy of both approaches (Q5), Table 7.2 shows the deviations between the predicted mean response times per usage profile (UP) of Palladio/SPE and the experiment's sample solution. The values are averaged among all design alternatives and all participants and therefore highly condensed for quick comprehension. More detail about the deviation can be found in [Mar07].

	Media Store		Web Server		Average
	$UP1$	$UP2$	$UP1$	$UP2$	
Palladio	4.69%	6.79%	7.45%	10.67%	6.9%
SPE	11.35%	10.21%	2.42%	9.21%	8.3%

Table 7.2: Deviation of the predicted response times

The maximum deviation is below 30%, therefore the hypothesis stated for Q5 cannot be rejected in this case. There are also no significant differences between Palladio and SPE as well as between both usage profiles. Notice that these values refer to predictions that passed the acceptances tests, other predictions are excluded. Thus, a limited deviation between the values was already ensured by the experiment's design.

There are many possible reasons for the still visible deviations. The simulations performed during the experiment do not produce deterministic results after each run, because they involve drawing samples from probability distributions. Furthermore, the participants may have not correctly used all values for performance annotations from the experiment's task description in their model. Some of these values had to be determined via additional calculations, which could introduce errors.

Question Q6 asked whether the participants could successfully rank the design alternatives according to their performance properties. Answering this questions turned out to be difficult, because several students did not complete the experimental tasks in the allotted time and could therefore not provide a full ranking. Table 7.3 shows the correct rankings of design alternatives in the form x/y, where x is the number of correct rankings and y is the number of students, who submitted a ranking at all.

	Media Store		Web Server	
	$UP1$	$UP2$	$UP1$	$UP2$
Palladio	1/1	1/1	4/6	5/5
SPE	7/7	2/7	8/8	7/8

Table 7.3: Correct Rankings of the Design Alternatives

Although the number of completed rankings is low (especially for Palladio), most submitted rankings were indeed correct. For the MediaStore and Palladio, only a single participant provided a ranking. The numbers are slightly higher for the WebServer. This might result from a learning effect of the participants, because they analysed the WebServer after the MediaStore and were more familiar with the experimental setting. Only 2 of 7 rankings for analysing usage profile 2 of the MediaStore turned out to match the sample solution. A reason might be the fact that for this setting the results for several design alternatives (e.g., i) and iv) as well as ii) and iii)) were almost equal, which easily lead to different rankings.

In conclusion the study provided too few data points to answer question Q6 reasonably. Therefore, the assessment of different design alternatives still remains to be validated properly.

Question Q7 asked for causes for the achieved prediction quality. Therefore, Martens [Mar07] analysed the problems and errors made by the participants during the experiment. The experimenters had documented all questions by the participants and the reasons for failed acceptance tests during the experiment. After the experiment, the student's models were analysed further for errors and deviations from the experimental tasks, which could not be revealed with the acceptance tests. Furthermore, the experimenters handed out questionnaires to the students, where they could themselves note problems they encountered.

For Palladio, many documented problems and errors related to the specification of parameter dependencies and component parameters. The number of problems

was higher in the first session when the MediaStore was analysed and less pronounced in the second session (for details see [Mar07]).

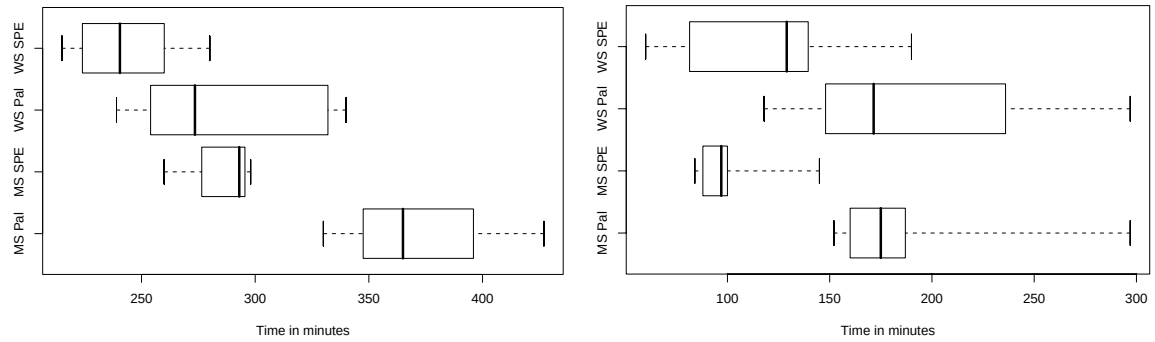
There are several possible explanations for the students' problems with specifying parameter dependencies. During the preparation phase, unfortunately, component parameters had only been introduced in the lectures, but not practiced in the exercises. The PCM-Bench's support for specifying parameter dependencies was still prototypical and not substantially tested. The concept of parameter dependencies is one of the few points, where the Palladio method differs substantially from the SPE method. Therefore, the participants could not rely on their SPE knowledge for specifying parameter dependencies, which they could for example for specifying control flow.

During the second session, fewer problems with parameter dependencies were detected, which might relate to a learning effect (more familiarity with the experimental setting) or to unbalanced groups. On the subjective questionnaires, the participants stated that the specification of parameter dependencies was time-consuming and error-prone with the available tools and that it reduced the clear overview of the complete model. However, they also noted that it lead to higher flexibility, enabled reuse, and was more intuitive than in the SPE method, where the influence of parameters on the performance had to be encoded into constant resource demands.

As another interesting fact, the problems with Palladio occurred earlier during the experiment and the finished Palladio models were less erroneous than the SPE models. 77% of Palladio-related problems occurred during modelling and were noted on the experimenters' questions protocols, 12% were encountered in the acceptance tests, and 11% were still present in the finished models after the experiment. In contrast, SPE-related problems occurred in 30% of all cases during the experiment and were noted on the question protocols, 26% at the acceptance tests, and 44% when analysing the finished models. This could relate to the PCM-Bench requiring more correct inputs and checking more substantially for errors, whereas the SPE-ED tool is more robust to erroneous user input and does not check the models for validity.

To answer question Q8 about the duration for the performance prediction with both methods, Fig. 7.17 shows box-and-whisker plots for the time spent by the students in the experiment sessions. The left hand side (Fig. 7.17(a)) illustrates the overall duration, while the right hand side (Fig. 7.17(b)) illustrates the duration for modelling and performance prediction of only the initial system without any design

alternatives. While the box plots show that the time varied strongly between different participants, it is also visible that the prediction for Palladio took longer for both analysed systems.



(a) Prediction including all Design Alternatives (b) Prediction without Design Alternatives

Figure 7.17: Box plots of the overall time needed by the students in the experiment sessions

To quantify the ratio between the durations of Palladio and SPE, Table 7.4 includes the mean values for the execution times. The ratio ($\frac{d(Palladio)}{d(SPE)}$) between the duration for the overall prediction is 1.32 for the MediaStore and 1.17 for the WebServer. It is interesting that the ratio was higher for both system, when only looking at the duration for modelling the initial system without any design alternatives. With Palladio, the students were able to model and analyse the design alternatives in 171 minutes (MediaStore) or 94 minutes (WebServer), which is less than for SPE, where they needed 185 minutes for the MediaStore and 124 minutes for the Webserver. However, for Palladio not all participants were able to finish the analysis of all design alternatives.

	Prediction including all Design Alternatives			Prediction without Design Alternatives		
	MediaStore	WebServer	Avg	MediaStore	WebServer	Avg
Palladio	374 min	285 min	329.5 min	203 min	191 min	197 min
SPE	284 min	243 min	263.5 min	99 min	119 min	109 min
Ratio	1.32	1.17	1.25	2.05	1.61	1.81

Table 7.4: Duration for the Predictions (mean values, in minutes)

This result could be a hint that the reusability of Palladio models was advantageous over the SPE models when modelling the design alternatives. In SPE, the

students had to model the control flow through the system again to assess a different design alternative. Instead Palladio allowed to simply relocate a component to a different server or introducing a new component without the need to model the control flow of the other components again. However, the number of data values is not sufficient to derive general observations with statistical significance.

Fig. 7.18 shows a more detailed breakdown of the duration for modelling the initial system without design alternatives. For both methods, it took the students less time to read the experimental tasks for the WebServer, which might be the result of a learning effect after the first session where the MediaStore was analysed. Also for both methods, more time was needed to model the control flow of the WebServer than the of the MediaStore, which could hint at a more complex architecture of the WebServer.

Figure 7.18: Breakdown of the duration for analysing the original system

Another interesting detail is the time needed by the students to search for errors in their models after the tools rejected the models. The time was significantly higher for Palladio that for SPE with both systems. This could be a sign of the still immature tool support for Palladio, which sometimes provided less informative error messages, whereas SPE also accepted erroneous models. The absolute numbers of errors with both methods were comparable.

7.4.4 Threats to Validity

The following discusses the internal validity, the construct validity, and the external validity of the experiment described before.

Internal Validity The internal validity refers to the degree to which changes in the dependent variables (i.e., the resulting metrics) are indeed results of changing the independent variables (i.e., the performance prediction method) and not caused by interfering variables [WRH+00]. The experimenters have to control all relevant interfering variables as much as possible to ensure a high internal validity.

An important interfering variable in this experiment were the *different capabilities* of the students. Based on their individual talent and motivation, the quality of the performance predictions might have changed. To ensure that at least the dependent variables changed equally for both investigated method, the students were assigned to the two experiment groups according to their exercise paper results. Students were randomly chosen from the better and the worse half based on these results and then assigned to the experiment groups to balance the capabilities of both groups.

The *systems under analysis* might also influence the prediction results, as one method could be more suited for a given system. To trace back differences of the dependent variables to the analysed systems and not to the prediction method Palladio or SPE, a change-over trial experiment design has been chosen. The participants of the experiment analysed two systems, therefore differences in the predictions and the duration can to a certain extent be traced back to the systems.

A small *learning effect* visibly appeared in the results of the experiment. During the second experiment session, the same students were able to conduct the performance predictions faster and more participants completed the analysis of all design alternatives. This might result from the students' higher familiarity with the experimental session and also from some bug fixes to the PCM bench after the first experiment session.

Furthermore threatening the internal validity is a potential *bias* of both the experimenters and the students. The experimenters developed the Palladio method and implemented the PCM bench. Therefore, they were naturally interested in showing the benefits of their own method against the SPE method. This possibly subconscious bias might have been transduced to the students during the tutorial sessions. The students might thus have been biased towards or against Palladio, because they knew the authors. However, the subjective results found in the questionnaires revealed no strong favouring of the Palladio approach over the SPE approach.

Construct Validity The construct validity refers to the degree to which persons and settings used in an experiment represent the analysed, more general constructs well [WRH+00]. In this case, the goal of the experiment was to analyse the practicability of two performance prediction methods.

Both SPE and Palladio represent the construct of a performance prediction method. SPE is a mature approach with substantial documentation and several available industrial case studies. It does not target a specific type of software system and can be applied broadly. Palladio is less mature than SPE and has not been validated with industrial-size case studies. It targets component-based systems and requires the analysed systems to follow this paradigm.

Although the methods slightly differ in their target systems, both use annotated control flow diagrams to model the performance properties of the system under study. In this regard, both approaches are representative for the many other approaches, which use annotated UML diagrams and transform them into formal prediction models [BDIS04].

To represent the setting of a performance prediction well, different design alternatives were analysed during the experiment. This shall simulate a typical situation for a software architect when designing a new system. The proposed design alternatives represent common performance patterns (e.g., cache, replication, resource pool) [Smi02].

The construct validity is also threatened by using students to conduct the performance prediction, which might not represent the situation in practice, where experienced software engineers analyse performance properties. However, the students in this experiment had completed their pre-diploma and were substantially trained in the investigated methods. Thus, it is questionable whether higher experience will indeed help practitioners to achieve better accuracy in a performance prediction.

External Validity The external validity refers to the degree to which the results of a study can be generalised to other in particular practical situations [WRH+00]. The results of the experiment presented here indicate a high prediction accuracy, while assessing longer durations for applying the Palladio methods. It is debatable whether these results would still be valid if larger systems had been analysed.

The systems under study were small and consisted of less than 10 components. This might threaten the external validity if systems in practice would be created from substantially more components. However, component granularity is variable [SGM02], therefore the architecture of a system with a higher number of com-

ponents could be abstracted into less than 10 components by forming composite components.

In their complexity and size, the analysed systems are comparable to systems analysed in typical other performance prediction case studies [Smi02]. The systems in these other studies are often taken from real-life projects. It is still unknown, whether the Palladio editing and simulation tools would be able to handle systems with a substantially finer component cutting. If performance analysts have to make model abstractions for such systems to be analysable, the prediction accuracy might decrease. Therefore, an experimental evaluation of the Palladio method with more complex systems is future work.

7.4.5 Conclusions

The described experiment was the first, initial type-II validation of the PCM and its accompanying tools. Because of the limited number of participants, no statistical hypothesis testing could be performed. Therefore, the results only give preliminary hints on the success of the validation without statistical significance.

The students were able to achieve a decent prediction accuracy with the Palladio method, but needed more time than for the SPE method. However, they created reusable models, whereas SPE models are not componentised and therefore more difficult to reuse. Although a research prototype, the PCM-Bench was able to support the performance modelling and simulation process well to a large extent.

The parameter dependencies in RDSEFF and the component parameters introduced in this thesis still proved to be difficult to apply for the participants of the experiment. This might result from a lack of training and still weak tool-support for these features. However, the experiment also provided hints that the parameterisation was advantageous over monolithic models, because the students were able to analyse the design alternatives of both systems faster with Palladio than with SPE.

Future experiments should improve the number of data points to enable statistical hypothesis testing. It would be interesting to include performance engineers from the software industry in the study, instead of just relying on students. In the future, the systems under study should be real systems from practice to increase the study's external validity. In the formerly described experiment, it was not possible to compare Palladio to another component-based performance prediction approach as non of the existing approaches provides the needed tool support. Therefore, the benefits of role concept of the PCM still remain to be validated.

7.5 Summary

This chapter presented an experimental evaluation of the modelling languages and transformations proposed in this thesis. First, a type I validation was conducted, where the PCM was used to model a distributed, component-based system, the Media Store. Because of the newly introduced parameter dependencies in the PCM, it was possible to analyse the performance of this architecture under two different usage profiles. Using the model-transformations and performance solvers described in this thesis, the response time of the Media Store was predicted for both usage profiles. A comparison with measurements showed a deviation of response time predictions of less than 10 percent using point estimators. Additionally, a sensitivity analysis involving more than 50 simulation runs revealed that the Media Store architecture is most sensitive to the number of users concurrently accessing it. Finally, to complement the type I evaluation, this chapter reported on a type II evaluation of the PCM, which was conducted as a controlled experiment with 19 computer science students. It showed that the students were able to achieve a decent prediction accuracy using the models and prediction tools, but that the tools still need improvement before being practicable in an industrial context.

Chapter 8

Conclusions

8.1 Summary

This thesis has proposed several new modelling languages, which shall increase model-driven performance prediction accuracy for component-based software systems in scenarios where the performance depends on the usage profile. Furthermore, it presented two model transformations from the new languages into existing performance models. For these performance models, solvers based on numerical analysis and simulation are available to derive performance metrics such as response time, throughput, and resource utilisation. The whole approach proposed in this thesis was evaluated in a case study with a component-based system and a controlled experiment, where 19 students used the proposed modelling languages. The following will briefly summarise the contributions of this thesis.

The RDSEFF modelling language (Chapter 4.3) enables component developers to specify the performance of software components in relation to parameter values. An RDSEFF models the behaviour of a component service only in terms of resource demands (internal actions) and calls to required services (external actions), and therefore abstracts from component code. It allows control flow constructs, such as sequence, alternative, loop, and fork, as well as the specification of parameter dependencies to resource demands and calls to required services. This thesis gives the language Petri net semantics (Chapter 4.4).

RDSEFF instances allow breaking down a usage profile at system boundaries to the usage profiles of individual components, because using the language each component specifies how it propagates data through an architecture. Because of the parameterisation, RDSEFF models are well-suited for reuse in different contexts.

Chapter 4.3.4 discussed limitations of this language, which include the missing support for message-oriented communication, the negligence of component internal state, and the limited scopes for stochastic dependencies.

The PCM usage modelling language (Chapter 4.2) has been proposed to enable domain experts to specify user behaviour at system boundaries. It includes modelling workloads (user population or user arrival rate), parameterised calls to system services, waiting delays, and probabilistic behaviour with sequences, alternatives, and loops. Multiple usage scenarios running in parallel can be modelled.

The usage modelling language is a domain-specific language restricted to concepts known to a domain expert without IT background. It is more expressive than Markov usage models, because it allows arbitrarily distributed loop iterations and parameterisation of calls. Other than typical UML models for performance prediction, the language does not refer to hardware resources or components. Therefore system usage can be changed independently from the system structure. The language assumes (Chapter 4.2.3) that a single user does not initiate system calls in parallel.

This thesis has also proposed a new method of characterising component service parameters for performance prediction (Chapter 4.1). To express uncertainty about actual user behaviour during early development stages, the PCM parameter abstractions allow modelling parameter values with random variables. Because a parameter's influence on the performance of a system is in some cases better determined by parameter meta-data instead of the parameter values, the PCM parameter abstractions allow modelling different parameter meta-data, such as the byte size or the number of elements in a collection. The clear distinction between different meta-data is needed to allow independent usage of the resulting specifications.

The PCM parameter abstractions are accompanied by the StoEx-framework (Chapter 3.3.6), which allows boolean and arithmetic operations on the random variables characterising parameters. This is useful to specify parameter dependencies in RDSEFFs and PCM usage models. The parameter abstractions so far assume primitive, collection, and composite data types and do not support other parameters such as pointers or streams.

To enable performance predictions with models specified as instances of the newly introduced modelling languages, this thesis defines two transformations into performance models. Because each of these transformations requires solving the parameter dependencies inside a PCM instance, this step has been implemented as a separate model transformation ("Dependency-Solver"), which can be used by any

transformation from PCM instances to performance models.

The Dependency Solver propagates parameter characterisations specified by the domain expert in the usage model through all RDSEFF specifications in a fully specified PCM instance. This allows substituting the parameter references inside RDSEFFs and then solving the resulting stochastic expressions. The results are branch probabilities, loop iterations numbers, and resource demands for a specific usage context. The following transformations can directly map this information to queueing networks or Petri nets. Chapter 6.2.5 discusses the computational complexity of this transformation.

The first model transformation to the performance domain defined in this thesis is to Stochastic Regular Expressions (SRE). The Palladio research group has developed this model for quick estimation of response times in single-user scenarios. This model supports calculations involving the general distribution functions used in PCM instances and also provides its result as a general distribution function. This thesis has extended the model with a new loop concept, which allows arbitrary distributed number of iterations instead of the former geometrically distributed number of iterations, which had proved unrealistically im many settings. Chapter 6.3.6 describes the assumptions underlying this model: it does not support concurrency and assumes stochastically independent random variables for resource demands. The transformation to SREs has been implemented prototypically and the models has been validated in a case study (Chapter 7.3.3).

The second model transformation maps valid PCM instances to Layered Queueing Networks (LQN). This is a popular performance model for distributed systems, which is based on extended queueing networks. There are efficient solvers for LQN instances based on mean-values analysis or simulation. The model is substantially more complex than the SRE model and allows mapping all behavioural constructs of PCM instances. It supports concurrency and asynchronous communication. However, the model assumes exponentially distributed resource demands and provides performance metrics only as mean-values. Therefore, LQN performance predictions cannot exploit the general distribution functions supported by PCM instances and provide less expressive results. The transformation from PCM instances to LQNs has been implemented prototypically in Java and validated in a case study (Chapter 7.3.3).

Besides new modelling languages and transformations, this thesis also proposes a new process model for QoS-driven development of component-based software systems (Chapter 3.1). It is based on a process model for component-based systems

by Cheesman et al. [CD01]. The PCM is designed to support this process model. It explicitly includes the participation of different developer roles in the modelling process during system specification. The process model also introduces the idea of restricted domain-specific modelling language for different developer roles. In addition to similar process models, it explicitly includes the roles of the system deployer and domain expert in QoS analysis. An experimental validation of this process model requires substantial effort and remains future work.

As software components often already exist in code when designing a new component-based system, it is desirable to automatically generate the needed abstract performance models from code as far as possible. This thesis has proposed a hybrid approach including static and dynamic code analysis to derive RDSEFF instances from Java code (Chapter 5). The static code analysis part has been implemented by a Master student [KKKR08] and has been validated on a component-based system. Given arbitrary Java code and component interfaces, the static analysis automatically performs the RDSEFF abstractions on source code. It can identify resource accesses but cannot derive resource demands from source, which requires dynamic analysis. In general, the static analysis can determine loop iteration number and parameter dependencies only in restricted cases due to the halting problem.

To validate the newly proposed modelling languages, several empirical studies have been conducted in the context of this thesis (Chapter 7). This thesis reports in detail on the last and most complex study involving the so-called MediaStore system. It is a component-based 3-tier architecture, which is completely modelled using the PCM. This enabled analysing performance critical use cases of the system using the introduced transformations and performance solvers (Chapter 7.3.3).

Additionally, the MediaStore system was implemented, which allowed comparing prediction based on the models with measurements based on the implementation and assessing the prediction errors. In the investigated cases, the error was below 10 percent when comparing point estimators, which is comparable to the prediction accuracy related studies (e.g., [LFG05]) and often sufficient to assess different design alternatives. However, the new modelling languages allow analysing different usage profiles without changing the component performance specifications, which was not or only limitedly possible with former approaches.

The ability to evaluate different design alternatives with the PCM was validated in former studies [KHB06]. In addition, this thesis includes a sensitivity analysis of the MediaStore system based on multiple simulation runs. It analyses different parameter ranges of the MediaStore and identifies the parameter the prediction re-

sults are most sensitive to (the number of users). The analysis is still restricted as it changes only a single parameter value at a time and does not show the effect of changing multiple parameters simultaneously.

Finally, a controlled experiment has been conducted by a Master student to validate the applicability of the PCM by third parties [Mar07]. This validation complements the formerly conducted empirical studies. 19 computer science students participated in the experiment and predicted the performance of various systems using the PCM tools after extensive training. The experiment showed that the students could achieve a decent prediction accuracy (less than 10 percent deviation to the sample solution). However, it also showed that the students still had difficulties to specify the PCM's parameter dependencies manually (Chapter 7.4). The modelling tools proved to be not sufficiently robust to modelling errors regarding parameter dependencies and need further improvement to achieve industrial maturity.

8.2 Benefits

The formerly described contributions shall improve model-driven performance predictions for component-based software systems. They target component developers, software architects, system deployers, and domain experts.

Component developers can specify the performance of their software components reflecting all influence factors not under their control. In this thesis especially the influence of different usage profiles was included. Availability of such specifications may improve the saleability of the components. Software architects may prefer purchasing and using such components, because they make the behaviour of their designed systems more predictable in advance. Opposed to monolithic model-driven performance prediction approaches, the component developer's specifications are reusable by different software architects in different contexts, because of their parameterisation. This might also increase the reuse of the component.

The static code analysis developed in this thesis eases the effort for component developers to provide performance specifications. Tools can already perform some of the abstractions needed for the models, which might even enable inexperienced developers to create performance specifications. The modelling languages provide a standard abstraction level and give developers a vocabulary to describe performance properties.

Software architects benefit from the proposed method, because they can assess the performance of their systems during early development stages. This can help

identifying bottlenecks and design flaws. Adapting the design early can reduce the costs for re-designing and re-implementing a system due to poor performance after implementation. Software architects can evaluate individual components and select among functional equivalent components with different performance properties. The costs for modelling are reduced for software architects, because the work for modelling is shared with the component developers.

In particular, using the modelling languages proposed in this thesis, software architects can easily analyse the performance of their architectural designs under different usage profiles. This enables quick assessment for different customer requirements. It also helps to renegotiate performance goals with customers, if a system design cannot fulfill the contractually specified requirements. Software architects can predict their performance for different user populations or parameter values. Thus, they can determine achievable performance goals.

Software architects can also assess different design alternatives. For the same functional requirements they can test different designs and their impact on the performance. With the predictions, they get quantitative data for decision support and do not have to rely on intuition only. If none of the available component can meet certain performance goals as predicted by the method, a design alternative can also be the implementation of a new software component. Therefore, performance prediction also supports make-or-buy decisions.

Using the Palladio method, software architects only need limited performance engineering knowledge, because they use only domain-specific modelling languages referring to concepts from their domain. The performance models and their analytical solver or simulation tools are encapsulated and do not have to be understood by the software architect. Therefore, even non-experts can possibly conduct performance predictions, if the described specifications are available.

8.3 Future Work

The following provides pointers for research extending the work conducted in this thesis. It is divided into short-term and long-term future work.

8.3.1 Short Term Future Work

Short term future work is mainly concerned with weakening the formerly discussed assumptions underlying the modelling languages and transformations. Further-

more, in the short-term existing prediction methods and modelling approaches could be connected to the PCM to further exploit the possibilities of usage profile propagation in other domains.

- **RDSEFF Behavioural Extensions:** Several limitations of RDSEFFs have been listed in Chapter 4.3.4. The RDSEFF's support for stochastic dependencies of random variables needs improvement, which would allow more accurate predictions. Furthermore, it is difficult to model component internal concurrency with RDSEFFs, especially if the forked threads involve performance-relevant parameter dependencies. Specifying asynchronous communication is possible with RDSEFFs using fork actions, however it is intricate and a construct for asynchronous external call actions would be desirable. Also a construct for reusing behaviour inside a single component by different RDSEFFs would ease their creation.

- **Reliability Prediction with RDSEFFs:** Former research by Reussner et al. [RSP03] has dealt with reliability prediction for component-based software architectures using a similar but more restricted notation to RDSEFFs. It is desirable to connect this work to the current PCM meta-model, which would require only small extensions to internal actions of RDSEFFs. The corresponding reliability solver could be used for PCM instances in this case, which would enable its further development. This step is also a prerequisite to performability analysis of PCM instances. However, the validation of reliability prediction models is still difficult.

- **Parameter Model Extensions:** The parameter model introduced in this thesis allows primitive, collection, and composite data types. This assumes that components always exchange complete data packages. However, some components (e.g., for multimedia content) communicate via streams (e.g., audio/video streams), and the PCM provides no support for modelling such communication. Furthermore, as discussed in Chapter 4.1.4, it is possible to implement algorithms traversing PCM `System` instances, which propagate parameter domain divisions due to control flow branches back to the PCM `Usage Model`. This would give the domain expert a default set of subdomains for certain parameters, for which only the probabilities would have to be specified. Finally, the Dependency Solver still has limited support for the characterisation of `INNER` collection elements (Chapter 4.1.3) and needs to be extended accordingly.

- **Textual Syntax for RDSEFFs:** The controlled experiment evaluating the PCM showed that a lot of the participating students had difficulties modelling the parameter dependencies introduced in this thesis with the graphical editors. An alternative to these editors would be a textual syntax for RDSEFFs and usage models describing the models like pseudo code, which is more familiar to many developers. This could ease the specification of parameter dependencies and also enable using features like auto completions and syntax highlighting. The textual specification can be more compact than the graphical notation, therefore possibly providing a better overview of complex models.

- **Extend Reverse Engineering Method:** Java2PCM requires the definition of a list of component interfaces to analyse source code and only supports static analysis. As Chouambe [Cho07] has implemented a component detection tool for arbitrary Java code called ArchiRec, which can identify component interfaces, it is desirable to combine both tools. Furthermore, Java2PCM's static analysis needs to be completed by a dynamic code analysis to determine resource demands. It requires a testbed for components, which generates stubs for required services, systematically generates parameter values for provided interfaces, and repeatedly executes the component with these values. Monitoring facilities of such a testbed must measure resource demands and collect the results in a database. An automatic analysis should be able to generate resource demand functions in dependency to input parameter from the data.

- **Transformations to Intermediate Models:** The model transformations implemented in this thesis do not use intermediate modelling languages between software models and performance models, such as CSM [PW06] or KLAPER [GMS05]. As transformation to different performance models exist from these languages, a mapping to them could connect those models and their performance solvers to the PCM. However, tool support is still immature for these intermediate languages and they still do not support some PCM concepts, which would therefore be lost for the transformation into performance models. Thus, mapping the PCM to these languages depends on their future development and tool support.

- **Integrate Specialised Solvers:** Verdickt et al. [VDTD07] proposed a method to combine a network simulator with an analytical LQN solver. This approach allows more refined predictions in network-intensive scenarios. The idea of

combining different solvers for higher prediction accuracy could be incorporated to the Palladio approach. It is possible to connect existing network simulators (e.g., ns-2 [Inf]) or other resource simulators to the Palladio approach. A special benefit of using Palladio instead of LQNs would be that Palladio components explicitly specify the amount of data they send over networks using parameter dependencies. This would allow for even more accurate predictions.

8.3.2 Long Term Future Work

Besides the formerly described short-term future work, there also some directions for long-term future work, which require more in-depth research. The following lists describes some of them:

- **Dynamic Architecture:** The PCM supports analysing static, component-based architectures with fixed connectors and fixed component allocations to resources. However, some systems (e.g., involving local mobility or web services) have a dynamic architecture, where component connectors can change, components can be allocated to different resources, and components can be replicated at runtime. Methods from Grassi et al. [GMS07a] and Caporuscio et al. [CMI07] allow performance prediction and analysis for dynamically reconfigurable architectures. Extending the PCM into this direction would increase the number of analysable systems.

- **Evaluation PCM Role Concept:** The QoS-driven process model proposed in this thesis includes a separation of the different developer roles involved in the creation of a component-based system, which is required to allow them to work independently from each other. This separation is also targeted by the modelling languages included in the PCM, which are designed for specific developer roles. It remains unclear, whether the separated specification of the performance-related information of component-based systems is indeed beneficial, or whether a single performance specialist should always conduct the analysis. To evaluate the role concept, a controlled experiment with a larger software architecture would be required, where the participants would embody certain developer roles. In the Type II evaluation presented in Chapter 7.4 each of the participants modelled all parts of the system, so that the experiment yielded more data points. Therefore, there was no division of work.

However, such an evaluation of the role concept is expensive and requires a high number of participants to deliver a sufficient amount of data points.

- **Integration of other QoS Attributes:** As mentioned before, the PCM could support the analysis of other compositional QoS attributes besides performance, such as reliability, availability, maintainability, and costs. Notice that some QoS attributes are not per se compositional, such as security and safety, because they lead to emergent properties of the system. A QoS attribute currently gaining importance is power consumption, which is often critical in large-scale software architectures and requires similar predictions as for performance or reliability. Besides analysing these QoS attributes in isolation, the PCM could also support their combined analysis, for example the performability combined from performance and reliability. As another example, Petriu et al. [PWP+07] analyse the impact of security features on the performance of a software system.

- **Internal State:** The internal state of a software component can influence its QoS characteristics in the same manner as input parameter [HMW04]. In the PCM, only a static abstraction of internal state is modelled with component parameters (Chapter 4.1.3), which cannot change during runtime. This abstraction avoid state space explosion, because a component-based system can have a huge number of user-dependent internal states. However, future extensions to the PCM could experiment with less high abstractions for the internal state and make it user-dependent and changeable in specific scenarios. For example, state machines could be used to model possible internal states and transitions between them. Component parameter could provide default values capturing the initial state of a component. It remains to be validated in which cases such a model is still solvable.

- **Dynamic Abstraction Levels:** A major problem in performance modelling is the unknown abstraction level for the models. In general, the most abstract models that still deliver performance results sufficient to support design decisions are desired. For component developers it is a priori often unknown, which features of their components influence the performance of the after architecture significantly and need to be modelled precisely. The software architect might want to create an abstract architectural model to retain mathematical tractability, but would have to rely on a complex component performance specifications from the component developers, who would be interested in

providing most accurate models. The level of abstraction necessary for an accurate performance model might also depend on the usage scenario. For example, in a scenario where response times are expected to be higher than 10 seconds, it might not be necessary to model features, which only change the performance in a range of milliseconds. Therefore, a tool-supported, dynamic abstraction of performance models (e.g., an automatic adaption of RDSEFFs) to the usage scenario would be desirable to increase the number of solvable models.

- **Automated Evaluation of Architectural Models** The PCM's feedback after running model solvers that deliver performance metrics is still limited. There are various possibilities for improvement: Tools could graphically highlight bottlenecks in a PCM model instance and annotate the predicted passage times for individuals actions into `System` (combined) and `RDSEFF` instances. Analysis tools could also be prepared for important recurring questions about the performance of a system, such as "what is the maximum throughput?", "what is the bottleneck resource?", etc., for which they would provide specialised visualisations. Smith et al. [Smi02] have documented several performance patterns and anti-patterns. It is conceivable to mine valid PCM instances for performance anti-patterns and automatically suggest solutions or even the introduction of a performance pattern (also see [CF07]). Performance solvers could also be adapted to allow an automatic multi-variable sensitivity analysis of the performance of a PCM instance. Bondarev et al. [BCdK07] proposes a method for automatically analysing different architectural alternatives and investigate multi-objective trade-offs. As the PCM targets analysing different design alternatives, more sophisticated support into this direction is desirable.

- **Model Libraries** The PCM aims at reuse of performance models. PCM component repositories and resource repositories support storing models for retrieval and reuse. For hardware resources and commonly used software components, it is useful to store PCM model instances in these repositories to build up model libraries, which software architects can exploit. This has also been proposed by Cortellessa et al. [CPR07]. The commercial company Hyperformix supplies a model library for thousands of different servers, so that performance analysts can analyse the performance on different hardware resources and answer sizing questions.

Appendix A

Contributions and Imported Concepts

The contributions of this thesis are embedded into the research of the Palladio group. Therefore, to delimit these contributions from contributions of other group members, the following two figures show the authors of different packages of the Palladio Component Model (PCM) and model transformations for the PCM, which produce other models or code.

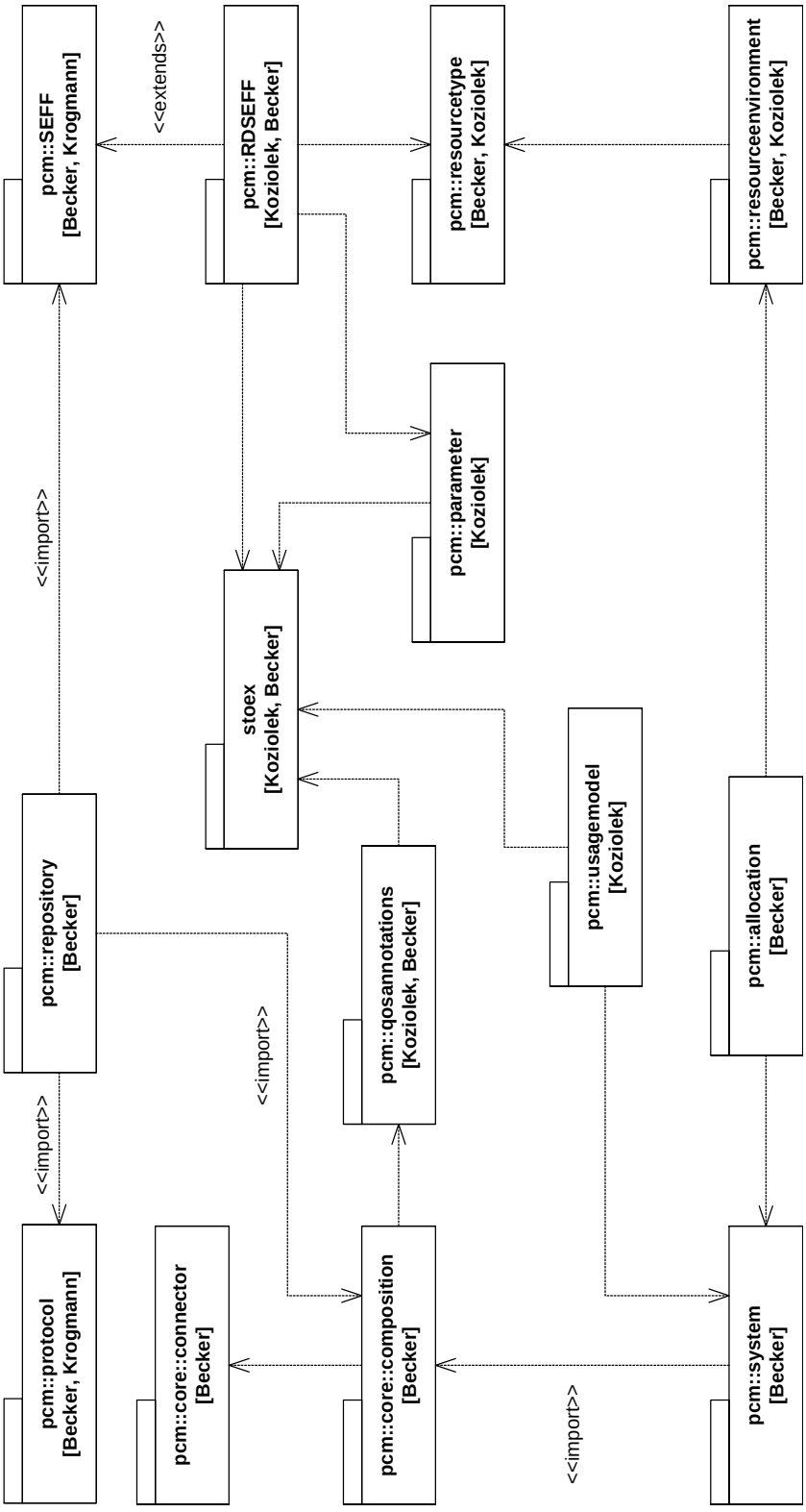


Figure A.1: Authors of PCM Packages

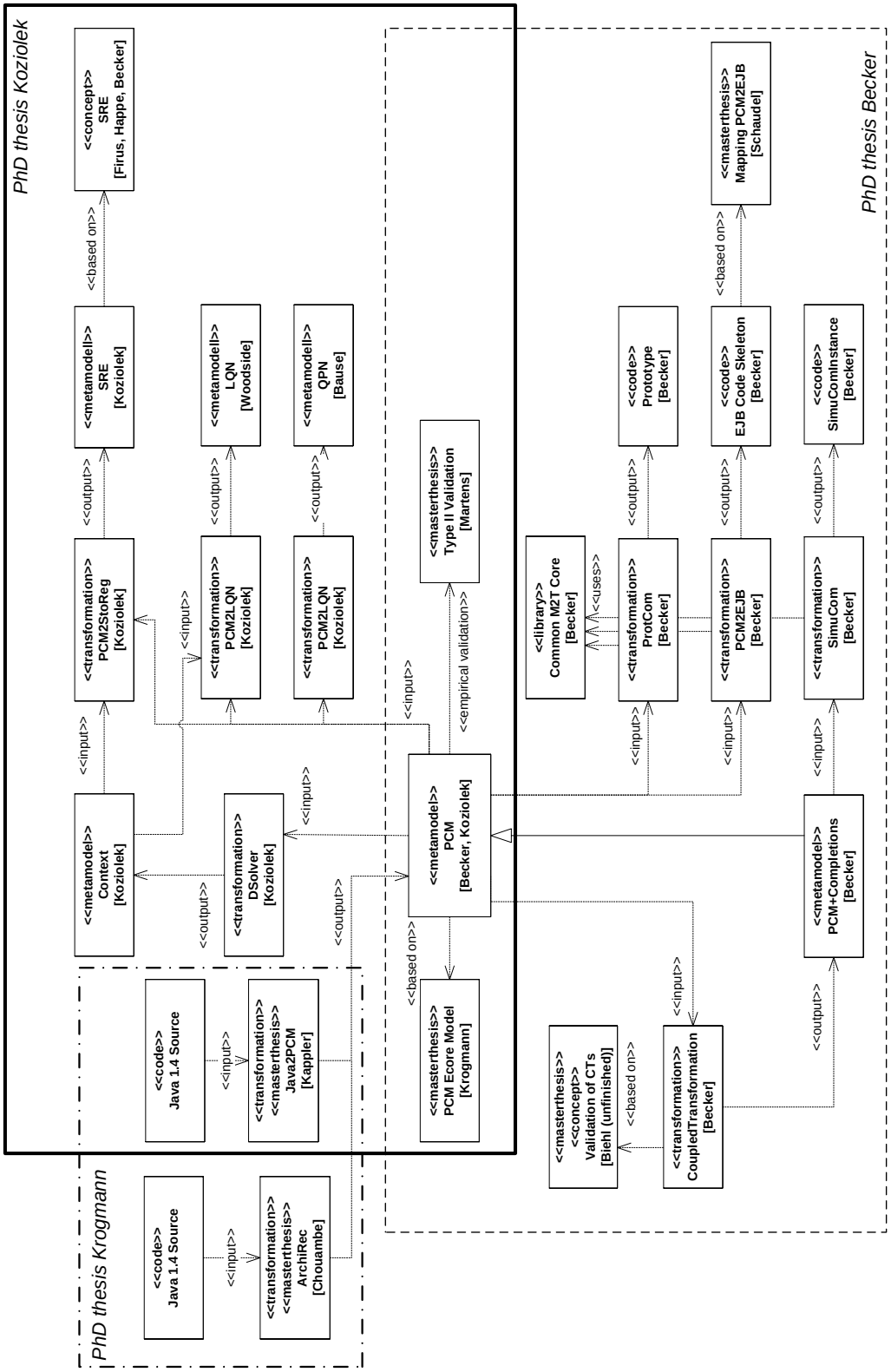


Figure A.2: Authors of PCM Transformations

Appendix B

Mathematical Definitions

B.1 Probability Theory

This section explains and formally defines some fundamental terms of probability theory.

B.1.1 Probability Space

The notion of probability space has been introduced by Kolmogorov in the 1930s and is the foundation of probability theory. Probability spaces are used to define random variables.

Definition 19 Probability Space

A ***proability space*** is a tupel $(\Omega, \mathcal{A}, P)$, with the pair $(\Omega, \mathcal{A})$ being a *measure space* consisting of

- the *sample space* Ω, which is a nonempty set, whose elements are given by the symbol ω and are known as outcomes of an experiment.
- the σ-algebra $\mathcal{A}$ of subsets of Ω, whose elements are called *events*. An event is a set of outcomes of an experiment to which a probability is assigned.

and the function $P : \mathcal{A} \to [0, 1]$ being a *probability measure* satisfying the following three probability axioms:

- $P(A) \geqslant 0$ for all $A \in \mathcal{A}$ (the probability of an event is a non-negative number)
- $P(\Omega) = 1$ (the probability that some elementary event in the entire sample space will occur is 1, i.e., there are no elementary events outside the sample space)
- $P(\bigcup_{i=1}^{\infty} A_i) = \sum_{i=1}^{\infty} P(A_i)$ for pairwise disjoint $A_1, A_2, ... \in \mathcal{A}$ (P is σ-additive)

B.1.2 Measurable Functions

Measurable functions are well-behaved functions between measure spaces. Non-measurable functions are generally considered pathological. A random variable variable is a measurable function, hence the definition follows.

Definition 20 Measurable Function

Let $(\Omega, \mathcal{A})$ and $(\Omega', \mathcal{A}')$ be measure spaces. The function $f : \Omega \to \Omega'$ is called **$\mathcal{A}$,$\mathcal{A}'$-measurable**, if

$$\forall A' \in \mathcal{A}' : f^{-1}(A') \in \mathcal{A}$$

B.1.3 Random Variable

The following, general definition of a random variable is not restricted to a specific measure space. Notice however, that normally (and also in the StoEx-framework) real-valued random variables are used.

Definition 21 Random Variable

Let $(\Omega, \mathcal{A}, P)$ be a probability space and $(\Omega', \mathcal{A}')$ be a measure space. An $(\mathcal{A}, \mathcal{A}')$-measurable function $X : \Omega \to \Omega'$ is called Ω'-***random variable*** on Ω.

B.1.4 Real-valued Random Variable

Definition 22 Real-valued Random Variable

Let $(\Omega, \mathcal{A}, P)$ be a probability space and $(\mathbb{R}, \mathcal{A}')$ be a measure space with $\mathcal{A}'$ being the Borel σ-algebra. An $(\mathcal{A}, \mathcal{A}')$-measurable function $X : \Omega \to \mathbb{R}$ is a ***real-valued random variable*** mapping a real number $X(\omega)$ to each element $\omega \in \Omega$, if

$$\forall r \in \mathbb{R} : \{\omega | X(\omega) \leqslant r\} \in \mathcal{A}$$

meaning that the set of all results below a certain value must be an event.

B.1.5 Probability Mass Function

A probability mass function gives the probability that a discrete random variable (with a finite or countable infinite sample space) is exactly equal to some value.

Definition 23 Probability Mass Function

Let $(\Omega, \mathcal{A}, P)$ be a probability space and X be a discrete random variable taking values on a countable sample space $S \subset \mathbb{R}$. Then the probability mass function $f_X : \mathbb{R} \to [0, 1]$ of X is given by

$$f_X(x) = \begin{cases} P(X = x), & x \in S \\ 0, & x \in \mathbb{R} \backslash S \end{cases}$$

This defines $f_X(x)$ for all real values, including the ones, x can never adopt. Their probability is always zero.

B.1.6 Probability Density Function

Definition 24 Probability Density Function

A non-negative Lebesgue-integrable function $f : \mathbb{R} \to \mathbb{R}$ is called ***probability density function*** of the random variable X, if

$$\int_a^b f(x)dx = P(a \leqslant X \leqslant b)$$

for any two numbers a and b, $a < b$. The total integral of $f(x)$ has to be 1 (i.e., $\int_{-\infty}^{\infty} f(x)dx = 1$).

B.2 Petri-Nets

This work uses a mapping to Hierarchical Queueing Petri nets (HQPN) to define the dynamic semantics of the behavioural models (Usage Model, RDSEFF) described in Chapter 4.2-4.3. HQPNs include a number of extensions to conventional Petri nets, which will be explained step by step in the following. The definition of Queuing Petri nets is taken from [BK02], while the definition of Hierarchical Peti nets follows [Jen92].

Definition 25 Petri Net [BK02]

An ordinary Petri Net (PN) is a 5-tuple $PN = (P, T, I^-, I^+, M_0)$, where

1. $P = \{p_1, p_2, ..., p_n\}$ is a finite and nonempty set of places,
2. $T = \{t_1, t_2, ..., t_m\}$ is a finite and nonempty set of transition $P \cap T = \varnothing$,
3. I^- and $I^+ : P \times T \to \mathbb{N}_0$ are called backward and forward incidence functions, respectively,
4. $M_0 : P \to \mathbb{N}_0$ is called initial marking.

Ordinary PNs cannot distinguish between different token types, which may lead to complicated models to express certain settings. Colored PN (CPN) allow to attach a type (called *color*) to each token. Each place is restricted to a set of colors, which specifies the valid types of tokens allowed to reside in it. Therefore, a color function C maps a set of colors to each place. Furthermore, the transitions of CPNs may fire in different *modes*, for which the color function C assigns a set of modes to each transition. CPNs are formally defined as follows:

Temporal aspects can be included into PNs with Stochastic PNs (SPN). SPN attach an exponentially distributed firing delay to each transition. This delay defines the time a transition waits after being enabled before it fires. Besides these timed transitions, Generalised Stochastic PNs (GSPN) additionally allow immediate transitions, which fire in zero time once enabled. Firing weights (i.e., probabilities) assigned to immediate transitions can be used to determine the next transitions to fire, if multiple immediate transitions are enabled. Immediate transition are prioritised over timed transitions and always fire in case a timed transition and an immediate transitioned are enabled at the same time. GSPN are formally defined as follows:

The last two definitions can be combined to define Colored GSPNs:

Definition 26 Colored Petri Net [BK02]

A Colored PN (CPN) is a 6-tuple $PN = (P, T, C, I^-, I^+, M_0)$, where

1. $P = \{p_1, p_2, ..., p_n\}$ is a finite and nonempty set of places,
2. $T = \{t_1, t_2, ..., t_m\}$ is a finite and nonempty set of transition $P \cap T = \varnothing$,
3. C is a color function that assigns a finite and nonempty set of colors to each place and a finite and nonempty set of modes to each transition.
4. I^-, I^+ are the backward and forward incidence functions defined on $P \times T$ such that
$$I^-(p,t), I^+(p,t) \in [C(t) \to C(p)_{MS}],$$
$\forall (p,t) \in P \times T$,
5. M_0 is a function defined on P describing the initial marking such that $M_0(p) \in C(p)_{MS}$.

Definition 27 Generalised Stochastic Petri Net (GSPN) [BK02]

A Generalised SPN is a 4-tuple $GSPN = (PN, T_1, T_2, W)$, where

1. $PN = (P, T, I^-, I^+, M_0)$ is the underlying ordinary PN,
2. $T_1 \subseteq T$ is the set of timed transitions $T_1 \neq \varnothing$,
3. $T_2 \subset T$ is the set of immediate transitions, $T_1 \cap T_2 = \varnothing, T_1 \cup T_2 = T$,
4. $W = (w_1, ..., w_{|T|})$ is an array whose entry $w_i \in \mathbb{R}^+$ is a rate of a negative exponential distribution specifying the firing delay, if $t_i \in T_1$ or is a firing weigh specifying the relative firing frequency, if $t_i \in T_2$.

Definition 28 Colored GSPN (CGSPN) [BK02]

A Colored GSPN is a 4-tuple $CGSPN = (CPN, T_1, T_2, W)$, where

1. $CPN = (P, T, C, I^-, I^+, M_0)$ is the underlying CPN,
2. $T_1 \subseteq T$ is the set of timed transitions $T_1 \neq \varnothing$,
3. $T_2 \subset T$ is the set of immediate transitions, $T_1 \cap T_2 = \varnothing, T_1 \cup T_2 = T$,
4. $W = (w_1, ..., w_{|T|})$ is an array with $w_i \in [C(t_i) \rightarrow \mathbb{R}^+]$ such that $\forall c \in C(t_i) : w_i(c) \in \mathbb{R}^+$ is a rate of a negative exponential distribution specifying the firing delay due to color c_i if $t_i \in T_1$ or is a firing weight specifying the relative firing frequency due to c, if $t_i \in T_2$.

CGSPN are not able to express queueing disciplines. Therefore Bause et. al [Bau93] introduced Queueing PN (QPN), which base on CGSPNs and feature places with integrated queues. A *queueing place* (Fig. B.1) consists of a *queue* for tokens requesting service and a *depository* for tokens, which have completed their service at the queue. If a token is fired into a queueing place, it gets inserted into the queue according to the queue's scheduling policy. Tokens in a queue cannot enable the output transitions of the queueing place. When a token completes its service, it is inserted into the depository, from which it can enable output transitions.

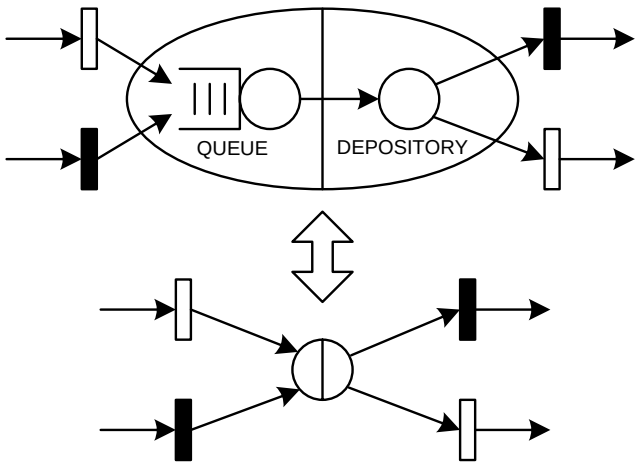


Figure B.1: Queueing Place and its Shorthand Notation [Kou06]

Besides *timed* queueing places, there are *immediate* queueing places for places

with zero service time. Analogous to immediate transitions, scheduling in immediate queueing places has priority over scheduling/service in time queueing places and firing of timed transitions. Otherwise, QPNs behave like CGSPNs, therefore the formal definition follows:

Definition 29 Queueing PN (QPN) [BK02]

A Queueing PN is a 8-tuple $QPN = (P, T, C, I^-, I^+, M_0, Q, W)$, where:

1. $CPN = (P, T, C, I^-, I^+, M_0)$ is the underlying CGSPN,
2. $Q = (\tilde{Q}_1, \tilde{Q}_2, (q_1, ..., q_{|P|}), (\mu_1, ..., \mu_{|P|}))$, where
   - $\tilde{Q}_1 \subseteq P$ is the set of timed queueing places,
   - $\tilde{Q}_2 \subseteq P$ is the set of immediate queueing places, $\tilde{Q}_1 \cap \tilde{Q}_2 = \varnothing$, and
   - q_i denotes the description of queue according to Kendall's notation [LZGS84] taking all colors of $C(p_i)$ into consideration, if p_i is a queueing place or equals the keyword "null" if p_i is an ordinary place,
   - $\mu_i \in [C(p_i) \to \mathbb{R}^+]$ such that $\forall c \in C(p_i) : \mu_i(c) \in \mathbb{R}^+$ is interpreted as a rate of a negative exponential distribution specifying the delay at place p_i due to color c.
3. $W = (\tilde{W}_1, \tilde{W}_2, (w_1, ..., w_{|T|}))$, where
   - $\tilde{W}_1 \subseteq T$ is the set of timed transitions,
   - $\tilde{W}_2 \subseteq T$ is the set of immediate transitions, $\tilde{W}_1 \cap \tilde{W}_2 = \varnothing, \tilde{W}_1 \cup \tilde{W}_2 = T$, and
   - $w_i \in [C(t_i) \to \mathbb{R}^+]$ such that $\forall c \in C(t_i) : w_i(c) \in \mathbb{R}^+$ is interpreted as a rate of a negative exponential distribution specifying the firing delay due to color c, if $t_i \in \tilde{W}_1$ or a firing weight specifying the relative firing frequency due to the color c, if $t_i \in \tilde{W}_2$.

It is convenient to structure larger PNs into smaller subnets. Therefore, Bause et al. [BBK94, BBK95] have introduced Hierarchical QPNs (HQPN), which consist of a number of QPN subnets and additionally contain *subnet places* (Fig.B.2). Each subnet has a dedicated input and output place and another place counting the active population of the subnet, which is the number of tokens fired into the subnet that have not yet left the subnet again.

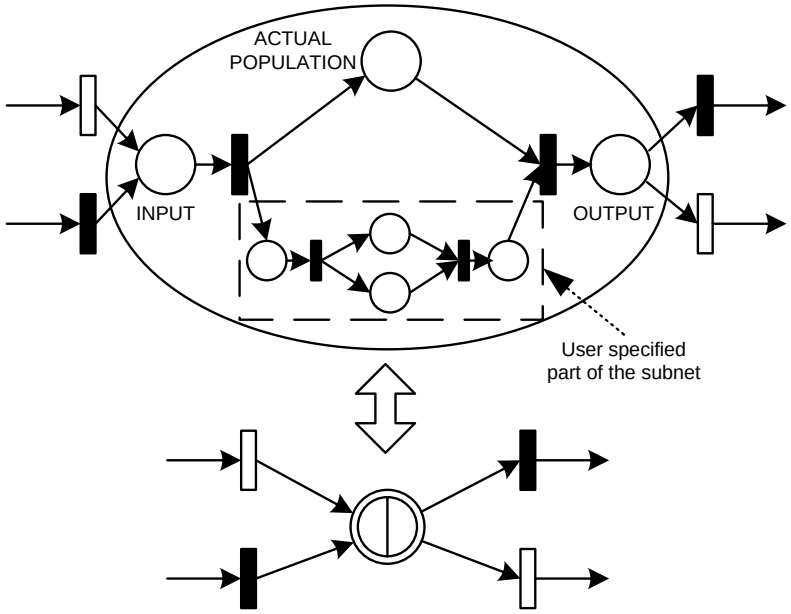


Figure B.2: Subnet Place and its Shorthand Notation [BBK94]

Tokens fired into a subnet place are added to the input place of the subnet, which the subnet place is assigned to. The semantics of the output places of subnets is similar to the depository of timed queuing places. Tokens added to a output place may enable output transitions of the corresponding subnet place.

Subnet places allow to create arbitrary hierarchies of QPNs. Note, that if two subnet places are assigned to the same subnet, an distinct instance of the subnet is created for each of them, i.e., tokens fired into both subnet places concurrently do not interfere with each other in a single subnet.

Jensen et al. [Jen92, p.97] have introduced another mechanism for structuring large PNs, which is called *fusion of places*. This mechanism allows users to specify that a set of places is considered to be identical. Whenever a token is added/removed at one of the places in a *fusion set*, an identical token is added/removed at all other places in the fusion set.

The following formal definition of HQPN includes both the subnet places [BBK94] and the fusion sets [Jen92] described above:

Definition 30 Hierarchical Queueing PN (HQPN) [BBK94, Jen92]

A Hierarchical Queueing PN is a 4-tuple $HQPN = (N, SP, SA, FS)$, where:

1. N is a finite set, where
   - $n \in N$ is a non hierarchical QPN $(P_n, T_n, C_n, I_n^-, I_n^+, M_{n_0}, Q_n, W_n)$,
   - the sets of net elements are pairwise disjoint: $\forall n_1, n_2 \in N : [n_1 \neq n_2 \Rightarrow (P_{n_1} \cup T_{n_1}) \cap (P_{n_2} \cup T_{n_2}) = \varnothing]$
2. $SP \subset P_N$ is the set of subnet places,
3. $SA : SP \rightarrow N$ is the subnet assignment function,
4. $FS \subseteq \mathcal{P}(P_N)$ is the set of fusion sets, such that members of a fusion set have identical colour sets and equivalent initialization expressions:
   - $\forall fs \in FS : \forall p_1, p_2 \in fs : [C(p_1) = C(p_2) \wedge M_0(p_1) = M_0(p_2)]$.

B.3 Scheduling Disciplines

- **Processor Sharing (PS):** An idealised form of round-robin scheduling with zero context switch times and zero time slices.
- **First-Come-First-Serve (FCFS):** Requests are served in the order in which they arrive.
- **Preemptive Priority Scheduling (PPS):** Requests with higher priorities will be served first. A request with a priority higher than the currently executing request will preempt the current request.
- **Head-of-Line Priority (HOL):** Requests with higher priorities will be served first. A request with a priority higher than the currently executing request will *not* preempt the current request.
- **Random Scheduling (RAND):** Requests will be selected for execution randomly.
- **Infinite Server (IS/INF):** An infinite number of servers for requests is assumed. This implies that each request will be served immediately without queueing.

Bibliography

[Aag01] Jan Aagedal. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, University of Oslo, 2001.

[App] Apple Incorporated. iTunes Music Store. `http://www.apple.com/itunes/`. last retrieved 2008-01-13.

[AW95] Alberto Avritzer and Elaine J. Weyuker. The automatic generation of load test suites and the assessment of the resulting software. *IEEE Trans. Softw. Eng.*, 21(9):705–716, 1995.

[Bau93] F. Bause. Queueing Petri Nets - A formalism for the combined qualitative and quantitative analysis of systems. In *Proceedings of the 5th International Workshop on Petri Nets and Performance Models*, pages 14–23, October 1993.

[BBG97] M. Bravetti, M. Bernardo, and R. Gorrieri. From EMPA to GSMPA: Allowing for General Distributions. In E. Brinksma and A. Nymeyer, editors, *Proc. of the 5th Int. Workshop on Process Algebras and Performance Modeling (PAPM'97)*, 1997.

[BBK94] F. Bause, P. Buchholz, and P. Kemper. Hierarchically combined queueing petri nets. In *Proceedings of the 11th International Conference on Analysis and Optimization of System*, volume 199 of *LNCS*, pages 176–182, Sophie-Anitpolis (France), June 1994.

[BBK95] Falko Bause, Peter Buchholz, and Peter Kemper. QPN-Tool for the specification and analysis of hierarchically combined queueing Petri nets. In *MMB '95: Proceedings of the 8th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, volume 977 of *Lecture Notes in Computer Science*, pages 224–238, London, UK, 1995. Springer-Verlag.

[BC96] R. J. A. Buhr and R. S. Casselman. *Use case maps for object-oriented systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[BCC+05] Jakob E. Bardram, Henrik Bærbak Christensen, Aino Vonge Corry, Klaus Marius Hansen, and Mads Ingstrup. Exploring quality attributes using architectural prototyping. In Ralf Reussner, Johannes Mayer, Judith A. Stafford, Sven Overhage, Steffen Becker, and Patrick J. Schroeder, editors, *Proc. 1st International Conference on the Quality of Software Architectures (QoSA'05)*, volume 3712 of *LNCS*, pages 155–170. Springer, 2005.

[BCdK07] Egor Bondarev, Michel R. V. Chaudron, and Erwin A. de Kock. Exploring performance trade-offs of a JPEG decoder using the deepcompass framework. In *Proc. of the 6th International Workshop on Software and performance (WOSP '07)*, pages 153–163, New York, NY, USA, 2007. ACM.

[BCdW06a] Egor Bondarev, Michel Chaudron, and Peter de With. A process for resolving performance trade-offs in component-based architectures. In *Proc. of the 9th International Symposium on Component-based Software Engineering (CBSE'06)*, volume 4063 of *LNCS*, pages 254–269. Springer, July 2006.

[BCdW06b] Egor Bondarev, Michel R. V. Chaudron, and Peter H. N. de With. Compositional performance analysis of component-based systems on heterogeneous multiprocessor platforms. In *Proc. 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO '06)*, pages 81–91. IEEE Computer Society, 2006.

[BDHK06] Henrik C. Bohnenkamp, Pedro R. D'Argenio, Holger Hermanns, and Joost-Pieter Katoen. MODEST: A compositional modeling formalism for hard and softly timed systems. *IEEE Trans. Software Eng*, 32(10):812–830, 2006.

[BDIS04] Simonetta Balsamo, Antinisca DiMarco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *IEEE Trans. Softw. Eng.*, 30(5):295–310, May 2004.

[BdWCM05] Egor Bondarev, Peter de With, Michel Chaudron, and Johan Musken. Modelling of Input-Parameter Dependency for Performance Predic-

tions of Component-Based Embedded Systems. In *Proc. of the 31th EUROMICRO Conference (EUROMICRO'05)*, 2005.

[Bec08] Steffen Becker. *Coupled Model Transformations for QoS Enabled Component-Based Software Design*. PhD thesis, University of Oldenburg, Germany, January 2008.

[BG98] M. Bernardo and R. Gorrieri. A tutorial on EMPA: A theory of concurrent processes with nondeterminism, priorities, probabilities and time. *Theoretical Computer Science*, 202(1-2):1–54, 1998.

[BG02] M. Bravetti and R. Gorrieri. The theory of interactive generalized semi-Markov processes. *Theoretical Computer Science*, 282(1):5–32, 2002.

[BGMO06] Steffen Becker, Lars Grunske, Raffaela Mirandola, and Sven Overhage. Performance Prediction of Component-Based Systems: A Survey from an Engineering Perspective. In Ralf Reussner, Judith Stafford, and Clemens Szyperski, editors, *Architecting Systems with Trustworthy Components*, volume 3938 of *LNCS*, pages 169–192. Springer, 2006.

[BH07] Marco Bernardo and Jane Hillston, editors. *Formal Methods for Performance Evaluation (7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM2007)*, volume 4486 of *LNCS*. Springer, May 2007.

[BJPW99] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7):38–45, 1999.

[BK02] F. Bause and F. Kritzinger. *Stochastic Petri Nets - An Introduction to the Theory*. Vieweg Verlag, 2nd edition, 2002.

[BKR95] Andrei Borshchev, Yuri Karpov, and Victor Roudakov. Covers - a tool for the design of real-time concurrent systems. In *Proc. 3rd International Conference on Parallel Computing Technologies (PaCT '95)*, pages 219–233, London, UK, 1995. Springer.

[BKR07] Steffen Becker, Heiko Koziolek, and Ralf Reussner. Model-based Performance Prediction with the Palladio Component Model. In *Proc. 6th International Workshop on Software and Performance (WOSP'07)*, pages 56–67. ACM Sigsoft, February 2007.

[BKR08] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The Palladio Component Model for Model-Driven Performance Prediction. *Journal of Systems and Software*, To appear:To appear, 2008.

[BM04a] Antonia Bertolino and Raffaela Mirandola. CB-SPE Tool: Putting component-based performance engineering into practice. In Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt C. Wallnau, editors, *Proc. 7th International Symposium on Component-Based Software Engineering (CBSE'04)*, volume 3054 of *LNCS*, pages 233–248. Springer, 2004.

[BM04b] Antonia Bertolino and Raffaela Mirandola. Software performance engineering of component-based systems. In *Proc. 4th International Workshop on Software and Performance (WOSP'04)*, pages 238–242, New York, NY, USA, 2004. ACM Press.

[BMdW+04] Egor Bondarev, Johan Muskens, Peter de With, Michel Chaudron, and Johan Lukkien. Predicting real-time properties of component assemblies: A scenario-simulation approach. In *Proc. 30th EUROMICRO Conference (EUROMICRO '04)*, pages 40–47. IEEE Computer Society, 2004.

[BMMI04] Simonetta Balsamo, Moreno Marzolla, Antinisca Di Marco, and Paola Inverardi. Experimenting different software architectures performance techniques: a case study. In *Proc. 4th International Workshop on Software and Performance (WOSP'04)*, pages 115–119, New York, NY, USA, 2004. ACM.

[BR06] S. Becker and R. Reussner. The Impact of Software Component Adaptation on Quality of Service Properties. *L'objet*, 12(1):105–125, 2006.

[BSM+03] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework*. Eclipse Series. Prentice Hall, August 2003.

[CBB+03] Paul C. Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures*. SEI Series in Software Engineering. Addison-Wesley, 2003.

[CD01] J. Cheesman and J. Daniels. *UML Components: A Simple Process for Specifying Component-based Software Systems*. Addison-Wesley, 2001.

[CDI01] V. Cortellessa, A. D'Ambrogio, and G. Iazeolla. Automatic derivation of software performance models from case documents. *Performance Evaluation*, 45(2-3):81–105, July 2001.

[CE00] Krysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CF07] Vittorio Cortellessa and Laurento Frittella. A framework for automated generation of architectural feedback from software performance analysis. In *Proc. 4th European Performance Engineering Workshop (EPEW'07)*, volume 4748 of *LNCS*, pages 171–185. Springer, September 2007.

[CGLL02] S. Chen, I. Gorton, A. Liu, and Y. Liu. Performance prediction of COTS component-based Enterprise Applications. In *Proc. 5th ICSE Workshop on Component-based Software Engineering (CBSE'02)*, 2002.

[CH06] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[Che80] Roger C. Cheung. A user-oriented software reliability model. *IEEE Trans. Softw. Eng.*, SE-6(2):118–125, March 1980.

[Cho07] Landry Chouambe. Rekonstruktion von Software-Architekturen. Master's thesis, Universität Karlsruhe (TH), May 2007.

[CLGL05] Shiping Chen, Yan Liu, Ian Gorton, and Anna Liu. Performance prediction of component-based applications. *J. Syst. Softw.*, 74(1):35–43, 2005.

[CMI07] Mauro Caporuscio, Antinisca Di Marco, and Paola Inverardi. Model-based system reconfiguration for dynamic performance management. *J. Syst. Softw.*, 80(4):455–473, 2007.

[CMZ02] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. Performance and scalability of EJB applications. *SIGPLAN Not.*, 37(11):246–261, 2002.

[CN82] K.M. Chandy and D. Neuse. Linearizer: a heuristic algorithm for queueing network models of computing systems. *Communications of the ACM*, 25(2):126–134, 1982.

[CN02] P. Clements and L. Northrop. *Software product lines*. Addison-Wesley Boston, 2002.

[Com06] Compuware. Applied performance management survey. `http://www.cnetdirectintl.com/direct/compuware/Ovum_APM/APM_Survey_Report.pdf`, October 2006. last retrieved 2008-01-13.

[Cor] Microsoft Corp. The COM homepage. `http://www.microsoft.com/com/`. last retrieved 2008-01-13.

[Cor05] Vittorio Cortellessa. How far are we from the definition of a common software performance ontology? In *Proc. 5th International Workshop on Software and Performance (WOSP '05)*, pages 195–204, New York, NY, USA, 2005. ACM Press.

[Cox86] Brad J. Cox. *Object oriented programming: an evolutionary approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[CPR07] Vittorio Cortellessa, Pierluigi Pierini, and Daniele Rossi. Integrating software models and platform models for performance analysis. *IEEE Trans. Softw. Eng.*, 33(6):385–401, June 2007.

[DFJR97] John Dilley, Rich Friedrich, Tai Jin, and Jerome A. Rolia. Measurement tools and modeling techniques for evaluating web server performance. In *Proceedings of the 9th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools*, pages 155–168, London, UK, 1997. Springer-Verlag.

[DG00] Karl Doerner and Walter J. Gutjahr. Representation and optimization of software usage models with non-markovian state transitions. *Information & Software Technology*, 42(12):873–887, Sep 2000.

[DI04] Antinisca DiMarco and Paola Inverardi. Compositional generation of software architecture performance qn models. In *Proc. 4th Working*

IEEE/IFIP Conference on Software Architecture (WISCA'04), pages 37–46. IEEE, June 2004.

[DiM05] Antinisca DiMarco. *Model-based Performance Analysis of Software Architectures*. PhD thesis, Universita di L'Aquila, 2005.

[DMM04] A. Diaconescu, A. Mos, and J. Murphy. Automatic performance management in component based software systems. In *Proc. IEEE International Conference on Autonomic Computing (ICAC'04)*, 2004.

[DPE04] Giovanni Denaro, Andrea Polini, and Wolfgang Emmerich. Early performance testing of distributed software applications. volume 29, pages 94–103, New York, NY, USA, 2004. ACM Press.

[Ecla] Eclipse Foundation. Atlas model weaver (amw). `http://www.eclipse.org/gmt/amw/`. last retrieved 2008-01-13.

[Eclb] Eclipse Foundation. Eclipse modeling framework homepage. `http://www.eclipse.org/modeling/emf/`. last retrieved 2008-01-13.

[Eclc] Eclipse Foundation. Graphical modeling framework homepage. `http://www.eclipse.org/gmf/`. last retrieved 2008-01-13.

[EF04] Evgeny Eskenazi and Alexander Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*. PhD thesis, Technische Universiteit Eindhoven, Netherlands, 2004.

[EFH04] Evgeni Eskenazi, Alexandre Fioukov, and Dieter Hammer. Performance Prediction for Component Compositions. In *Proc. 7th International Symposium on Component-based Software Engineering (CBSE'04)*, volume 3054 of *LNCS*. Springer, 2004.

[EHL+94] Stephen H. Edwards, Wayne D. Heym, Timothy J. Long, Murali Sitaraman, and Bruce W. Weide. Part ii: specifying components in resolve. *SIGSOFT Softw. Eng. Notes*, 19(4):29–39, 1994.

[EJB07] Sun Microsystems Corp., The Enterprise Java Beans homepage. `http://java.sun.com/products/ejb/`, 2007. last retrieved 2008-01-13.

[FBH05] Viktoria Firus, Steffen Becker, and Jens Happe. Parametric performance contracts for QML-specified software components. In *Proceedings of 2nd International Workshop on Formal Foundations of Embedded*

Software and Component-Based Software Architectures (FESCA '05), pages 64–79, 2005.

[FFO02] Andre G. Farina, Paulo Fernandes, and Flavio M. Oliveira. Representing software usage models with stochastic automata networks. In *Proc. 14th International Conference on Software Engineering and Knowledge Engineering (SEKE '02)*, pages 401–407, New York, NY, USA, 2002. ACM Press.

[FMN+96] G. Franks, S. Majumdar, J. Neilson, D. Petriu, J. Rolia, and M. Woodside. Performance analysis of distributed server systems. In *Proc. 6th International Conference on Software Quality (ICSQ'96)*, pages 15–26, 1996.

[FMW+07] Greg Franks, Peter Maly, Murray Woodside, Dorina Petriu, and Alex Hubbard. Layered queueing network solver and simulator user manual. `http://www.sce.carleton.ca/rads/lqns/LQNSUserMan.pdf`, May 2007. last retrieved 2008-01-13.

[FOL08] Free on-line dictionary of computing (foldoc). `http://foldoc.org/`, January 2008.

[Fra99] Greg Franks. *Performance Analysis of Distributed Server Systems*. PhD thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada, December 1999.

[FW98] Greg Franks and Murray Woodside. Performance of multi-level client-server systems with parallel service operations. In *Proc. 1st International Workshop on Software and Performance (WOSP'98)*, pages 120–130, New York, NY, USA, 1998. ACM.

[Gel] Jean Gelissen. Robocop: Robust open component based software architecture. `http://www.hitech-projects.com/euprojects/robocop/deliverables.htm`. last retrieved 2008-01-13.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GHR92] N. Goetz, U. Herzog, and M. Rettelbach. TIPP— a language for timed processes and performance evaluation. Technical report, 1992.

[GL03] Jean Gelissen and Ronan Mac Laverty. Robocop: Revised specification of framework and models (deliverable 1.5). Technical report, Information Technology for European Advancement, 2003.

[GLB04] Mechelle Gittens, Hanan Lutfiyya, and Michael Bauer. An extended operational profile model. In *Proc. 15th International Symposium on Software Reliability Engineering (ISSRE'04)*, pages 314–325, Washington, DC, USA, 2004. IEEE Computer Society.

[GM01] Hassan Gomaa and Daniel A. Menasce. Performance engineering of component-based distributed software systems. In *Performance Engineering, State of the Art and Current Trends*, volume 2047 of *LNCS*, pages 40–55, London, UK, 2001. Springer.

[GMS05] Vincenzo Grassi, Raffaela Mirandola, and Antonino Sabetta. From design to analysis models: a kernel language for performance and reliability analysis of component-based systems. In *Proc. 5th International Workshop on Software and Performance (WOSP '05)*, pages 25–36, New York, NY, USA, 2005. ACM Press.

[GMS07a] Vincenzo Grassi, Raffaela Mirandola, and Antonino Sabetta. A Model-Driven Approach to Performability Analysis of Dynamically Reconfigurable Component-Based Systems. In *Proc. 6th Workshop on Software and Performance (WOSP'07)*, February 2007.

[GMS07b] Vincenzo Grassi, Raffaela Mirandola, and Antonino Sabetta. Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach. *Journal on Systems and Software*, 80(4):528–558, 2007.

[GPT01] Katerina Goseva-Popstojanova and Kishor S. Trivedi. Architecture-based approach to reliability assessment of software systems. *Perform. Eval.*, 45(2-3):179–204, 2001.

[Gru07] Lars Grunske. Early quality prediction of component-based systems - a generic framework. *J. Syst. Softw.*, 80(5):678–686, 2007.

[GSCK04] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons, 2004.

[Gut00] Walter J. Gutjahr. Software dependability evaluation based on markov usage models. *Perform. Eval.*, 40(4):199–222, 2000.

[Hap04] Jens Happe. Reliability Prediction of Component-Based Software Architectures. Master's thesis, University of Oldenburg, 2004.

[Hap08] Jens Happe. *Concurrency Modelling for QoS-predictions of Software Components*. PhD thesis, University of Oldenburg, Germany, 2008. To Appear.

[HC01] George T. Heineman and William T. Councill, editors. *Component-based software engineering: putting the pieces together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Hil96] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[HKR06] Jens Happe, Heiko Koziolek, and Ralf Reussner. Parametric Performance Contracts for Software Components with Concurrent Behaviour. In Frank S. de Boer and Vladimir Mencl, editors, *Proc. 3rd International Workshop on Formal Aspects of Component Software (FACS'06)*, Electronical Notes in Computer Science, September 2006.

[HMSW02] Scott A. Hissam, Gabriel A. Moreno, Judith A. Stafford, and Kurt C. Wallnau. Packaging Predictable Assembly. In *Proc. IFIP/ACM Working Conference on Component Deployment (CD'02)*, pages 108–124, London, UK, 2002. Springer-Verlag.

[HMSW03] Scott Hissam, Gabriel Moreno, Judith Stafford, and Kurt Wallnau. Enabling predictable assembly. *J. Syst. Softw.*, 65(3):185–198, 2003.

[HMW01] Dick Hamlet, David Mason, and Denise Woit. Theory of software reliability based on components. In *Proc. 23rd International Conference on Software Engeneering (ICSE'01)*, pages 361–370, Los Alamitos, California, May12–19 2001. IEEE Computer Society.

[HMW04] Dick Hamlet, Dave Mason, and Denise Woit. *Properties of Software Systems Synthesized from Components*, volume 1 of *Series on Component-Based Software Development*, chapter Component-Based Software Development: Case Studies, pages 129–159. World Scientific Publishing Company, March 2004.

[hn07] heise newsticker. Bericht: Probleme bei SAPs neuer Mittelstandssoftware. http://www.heise.de/newsticker/meldung/88300, April 2007. last retrieved 2008-01-13.

[Hoa85] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[HRW95] Curtis E. Hrischuk, Jerome A. Rolia, and C. Murray Woodside. Automatic Generation of a Software Performance Model Using an Object-Oriented Prototype. In *Proc. 3rd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '95)*, pages 399–409, Washington, DC, USA, 1995. IEEE Computer Society.

[IEE00] IEEE. IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. Standard IEEE 1471-2000, 2000.

[Inf] Information Sciences Institute (ISI). Network Simulator ns-2. http://www.isi.edu/nsnam/ns/. last retrieved 2008-01-13.

[ISO03] ISO/IEC Standard. Software engineering – product quality – part 1: Quality model. ISO Standard 9126-1, ISO/IEC, 2003.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis : Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1991.

[Jen92] Kurt Jensen. *Coloured Petri Nets - Basic Concpets, Analysis Methods and Practical Use*, volume 1 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1992.

[Kan03] Michael Kanellos. Moore's Law to roll on for another decade. http://www.news.com/2100-1001-984051.html, February 2003. last retrieved 2008-01-13.

[Kap07] Thomas Kappler. Code Analysis Using Eclipse to Support Performance Predictions for Java Components. Master's thesis, Universität Karlsruhe (TH), August 2007.

[KB06] Samuel Kounev and Alejandro P. Buchmann. SimQPN - A tool and methodology for analyzing queueing petri net models by means of simulation. *Perform. Eval*, 63(4-5):364–394, 2006.

[KB07] Michael Kuperberg and Steffen Becker. Predicting Software Component Performance: On the Relevance of Parameters for Benchmarking Bytecode and APIs. In Ralf Reussner, Clemens Czyperski, and Wolfgang Weck, editors, *Proc. 12th International Workshop on Component Oriented Programming (WCOP'07)*, July 2007.

[KBH07] Heiko Koziolek, Steffen Becker, and Jens Happe. Predicting the Performance of Component-based Software Architectures with different Usage Profiles. In *Proc. 3rd International Conference on the Quality of Software Architectures (QoSA'07)*, volume 4880 of *LNCS*, pages 145–163. Springer, Juli 2007.

[KBHR08] Heiko Koziolek, Steffen Becker, Jens Happe, and Ralf Reussner. *Model-Driven Software Development: Integrating Quality Assurance*, chapter Evaluating Performance and Reliability of Software Architectures with the Palladio Component Model, page To appear. IDEA Group Inc., December 2008.

[KBWA94] Rick Kazman, Len Bass, Mike Webb, and Gregory Abowd. SAAM: a method for analyzing the properties of software architectures. In *Proc. 16th International Conference on Software Engineering (ICSE '94)*, pages 81–90, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[KF05] Heiko Koziolek and Viktoria Firus. Empirical Evaluation of Model-based Performance Predictions Methods in Software Development. In Ralf Reussner, Johannes Mayer, Judith A. Stafford, Sven Overhage, Steffen Becker, and Patrick J. Schroeder, editors, *Quality of Software Architectures and Software Quality (Proceedings of the First International Conference on Quality of Software Architectures (QoSA2005))*, volume 3712 of *Lecture Notes in Computer Science*, pages 188–202, Erfurt, Germany, 9 2005.

[KF06] Heiko Koziolek and Viktoria Firus. Parametric Performance Contracts: Non-Markovian Loop Modelling and an Experimental Evaluation. In Juliana Kuester-Filipe, Iman H. Poernomo, and Ralf Reussner, editors, *Proc. 3rd International Workshop on Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA'06)*, volume 176 of *ENTCS*, pages 69–87. Elsevier, March 2006.

[KH06] Heiko Koziolek and Jens Happe. A QoS-Driven Development Process Model for Component-Based Software Systems. In Ian Gorton, George T. Heineman, Ivica Crnkovic, Heinz W. Schmidt, Judith A. Stafford, Clemens A. Szyperski, and Kurt C. Wallnau, editors, *Proc. 9th International Symposium on Component-Based Software Engineering (CBSE'06)*, volume 4063 of *LNCS*, pages 336–343. Springer, June 2006.

[KHB06] Heiko Koziolek, Jens Happe, and Steffen Becker. Parameter Dependent Performance Specifications of Software Components. In Christine Hofmeister, Ivica Crnkovic, Ralf Reussner, and Steffen Becker, editors, *Proc. 2nd International Conference on the Quality of Software Architectures (QoSA'06)*, volume 4214 of *LNCS*, pages 163–179. Springer, June 2006.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[KKC00] R. Kazman, M. Klein, and P. Clements. Atam: Method for architecture evaluation. Technical Report CMU/SEI-2000-TR-004, Carnegie Mellon University, Software Engineering Institute, 2000.

[KKKR08] Thomas Kappler, Heiko Koziolek, Klaus Krogmann, and Ralf Reussner. Towards Automatic Construction of Reusable Prediction Models for Component-Based Performance Engineering. In *Proc. Software Engineering 2008 (SE'08)*, LNI. GI, February 2008. To Appear.

[Kos05] Rainer Koschke. Rekonstruktion von Software-Architekturen – Ein Literatur- und Methoden-Überblick zum Stand der Wissenschaft. *Informatik – Forschung und Entwicklung*, 19(3):127–140, April 2005. Springer Berlin / Heidelberg.

[Kou06] Samuel Kounev. Performance Modeling and Evaluation of Distributed Component-Based Systems Using Queueing Petri Nets. *IEEE Trans. Softw. Eng.*, 32(7):486–502, July 2006.

[Koz04] Heiko Koziolek. Empirische Bewertung von Performance-Analyseverfahren für Software-Architekturen. Diploma thesis, Universität Oldenburg, October 2004.

[Koz05] Heiko Koziolek. Operational profiles for software reliability. In Wilhelm Hasselbring and Simon Giesecke, editors, *Dependability Engineer-*

ing, volume 2 of *Trustworthy Software Systems*, chapter 6, pages 119–142. GITO-Verlag, Berlin, 2006, 2005.

[KPV03] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, Warsaw, Poland, April 2003.

[KR08] Klaus Krogmann and Ralf Reussner. Palladio - Prediction of Performance Properties. In *The Common Component Modeling Example: Comparing Software Component Models*,, To Appear in LNCS. Springer, 2008.

[Kro07] Klaus Krogmann. Reengineering of Software Component Models to Enable Architectural Quality of Service Predictions. In Ralf Reussner, Clemens Szyperski, and Wolfgang Weck, editors, *Proc. 12th International Workshop on Component Oriented Programming (WCOP'07)*, July 2007.

[KWB03] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[LFG05] Yan Liu, Alan Fekete, and Ian Gorton. Design-level performance prediction of component-based applications. *IEEE Trans. Softw. Eng.*, 31(11):928–941, 2005.

[LZ74] Barbara Liskov and Stephen Zilles. Programming with abstract data types. *SIGPLAN Not.*, 9(4):50–59, 1974.

[LZGS84] E.D. Lazowska, J. Zahorjan, G.S. Graham, and K.C. Sevcik. *Quantitative System Performance*. Prentice Hall, 1984.

[MAFM99] Daniel A. Menasce, Virgilio A. F. Almeida, Rodrigo Fonseca, and Marco A. Mendes. A methodology for workload characterization of e-commerce sites. In *EC '99: Proceedings of the 1st ACM conference on Electronic commerce*, pages 119–128, New York, NY, USA, 1999. ACM Press.

[Mar04] M. Marzolla. *Simulation-Based Performance Modeling of UML Software Architectures*. PhD thesis, Universit'a Ca Foscari di Venezia, 2004.

[Mar05] Anne Martens. Empirical Validation and Comparison of the Model-Driven Performance Prediction Techniques of CB-SPE and Palladio. Individuelles projekt, Universität Oldenburg, August 2005.

[Mar07] Anne Martens. Empirical Validation of the Model-driven Performance Prediction Approach Palladio. Master's thesis, University of Oldenburg, November 2007.

[MBH+04] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry R. Payne, Evren Sirin, Naveen Srinivasan, and Katia Sycara. Owl-s: Semantic markup for web services. `http://www.w3.org/Submission/2004/07/`, Novemberq 2004. last retrieved 2008-01-13.

[MBNR68] M.D. McIlroy, JM Buxton, P. Naur, and B. Randell. Mass-Produced Software Components. *Software Engineering Concepts and Techniques (NATO Science Committee)*, 1:88–98, 1968.

[McM01] David McMullan. Components in layered queueing networks. Technical report, Carlton University, Ottawa, Canada, October 2001.

[MDL87] HD Mills, M. Dyer, and RC Linger. Cleanroom Software Engineering. *IEEE Software*, 4(5):19–25, 1987.

[met07] metamodel.com. What is metamodeling, and what is it good for? `http://www.metamodel.com/staticpages/index.php?page=20021010231056977`, 2007. last retrieved 2008-01-13.

[Mey90] Bertrand Meyer. *Introduction to the theory of programming languages*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[Mey92] Bertrand Meyer. Applying Design by Contract. *Computer*, 25(10):40–51, 1992.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction, Second Edition*. The Object-Oriented Series. Prentice-Hall, Englewood Cliffs (NJ), USA, 1997.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[MJ51] F.J. Massey Jr. The Kolmogorov-Smirnov Test for Goodness of Fit. *Journal of the American Statistical Association*, 46(253):68–78, 1951.

[ML05] Marcus Meyerhöfer and Frank Lauterwald. Towards platform-independent component measurement. In *Proceedings of the 10th Workshop on Component-Oriented Programming (WCOP2005)*, 2005.

[MN04] Marcus Meyerhöfer and Christoph Neumann. TESTEJB - A Measurement Framework for EJBs. In *Proceedings of the 7th International Symposium on Component-Based Software Engineering (CBSE7)*, 2004.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[Mus93] John D. Musa. Operational profiles in software-reliability engineering. *IEEE Software*, 10(2):14–32, 1993.

[MV05] Marcus Meyerhöfer and Bernhard Volz. EJBMemprof - A memory profiling framework for enterprise javabeans. In *Proc. 8th International Symposium on Component-based Software Engineering (CBSE'05)*, volume 3489 of *LNCS*, pages 17–32, 2005.

[MW00] Peter Maly and C. Murray Woodside. Layered modeling of hardware and software, with application to a lan extension router. In *Proc. 11th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools (TOOLS'00)*, pages 10–24, London, UK, 2000. Springer-Verlag.

[Nei91] J.E. Neilson. *Parasol: A Simulator for Distributed And/or Parallel Systems*. Carleton University, School of Computer Science, 1991.

[NWPM95] J. E. Neilson, C. M. Woodside, D. C. Petriu, and S. Majumdar. Software bottlenecking in client-server systems and rendezvous networks. *IEEE Trans. Softw. Eng.*, 21(9):776–782, 1995.

[Obj05a] Object Management Group (OMG). UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms. `http://www.omg.org/cgi-bin/doc?ptc/2005-05-02`, May 2005. last retrieved 2008-01-13.

[Obj05b] Object Management Group (OMG). UML Profile for Schedulability, Performance and Time. http://www.omg.org/cgi-bin/doc?formal/2005-01-02, 2005. last retrieved 2008-01-13.

[Obj06a] Object Management Group (OMG). CORBA 3.0 - IDL Syntax and Semantics chapter. http://www.omg.org/cgi-bin/doc?formal/02-06-07, February 2006. last retrieved 2008-01-13.

[Obj06b] Object Management Group (OMG). Corba component model, v4.0 (formal/2006-04-01). http://www.omg.org/technology/documents/formal/components.htm, 2006. last retrieved 2008-01-13.

[Obj06c] Object Management Group (OMG). Metaobject facility (MOF). http://www.omg.org/mof/, 2006. last retrieved 2008-01-13.

[Obj06d] Object Management Group (OMG). MOF QVT final adopted specification (ptc/05-11-01). http://www.omg.org/cgi-bin/apps/doc?ptc/05-11-01.pdf, 2006. last retrieved 2008-01-13.

[Obj07a] Object Management Group (OMG). UML Profile for MARTE, Beta 1. http://www.omg.org/cgi-bin/doc?ptc/2007-08-04, August 2007. last retrieved 2008-01-13.

[Obj07b] Object Management Group (OMG). Unified modeling language: Superstructure version 2.1.1. http://www.omg.org/cgi-bin/doc?formal/07-02-05, February 2007. last retrieved 2008-01-13.

[OFWP05] Tariq Omari, Greg Franks, Murray Woodside, and Amy Pan. Solving layered queueing networks of large client-server systems with symmetric replication. In *WOSP '05: Proceedings of the 5th international workshop on Software and performance*, pages 159–166, New York, NY, USA, 2005. ACM.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[PPK06] Pavel Parizek, Frantisek Plasil, and Jan Kofron. Model checking of software components: Combining java pathfinder and behavior protocol model checker. In *Proceedings of the 30th Annual IEEE/NASA*

Software Engineering Workshop SEW-30 (SEW'06), pages 133–141. IEEE Computer Society, 2006.

[PS02] Dorina C. Petriu and Hui Shen. Applying the UML Performance Profile: Graph Grammar-Based Derivation of LQN Models from UML Specifications. In *Proc. 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools (TOOLS'02)*, pages 159–177, London, UK, 2002. Springer-Verlag.

[PTLP99] S.J. Prowell, C.J. Trammell, R.C. Linger, and J.H. Poore. *Cleanroom software engineering: technology and process*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.

[PW91] D.C. Petriu and C.M. Woodside. Approximate MVA from Markov model of software client/server systems. In *Proc. 3rd IEEE Symposium on Parallel and Distributed Processing (PDP'91)*, pages 322–329, 1991.

[PW02] Dorin C. Petriu and C. Murray Woodside. Software Performance Models from System Scenarios in Use Case Maps. In *Proc. 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools (TOOLS'02)*, pages 141–158, London, UK, 2002. Springer-Verlag.

[PW04] Dorin B. Petriu and Murray Woodside. A metamodel for generating performance models from UML designs. In Thomas Baar, Alfred Strohmeier, Ana Moreira, and Stephen J. Mellor, editors, *UML 2004 - The Unified Modeling Language. Model Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004, Proceedings*, volume 3273 of *LNCS*, pages 41–53. Springer, 2004.

[PW06] Dorin B. Petriu and Murray Woodside. An intermediate metamodel with scenarios and resources for generating performance models from UML designs. *Journal of Software and Systems Modeling*, 6(2):163–184, June 2006.

[PWP+07] D. C. Petriu, C. M. Woodside, D. B. Petriu, J. Xu, T. Israr, Geri Georg, Robert France, James M. Bieman, Siv Hilde Houmb, and Jan Jürjens. Performance analysis of security aspects in uml models. In *WOSP '07: Proceedings of the 6th international workshop on Software and performance*, pages 91–102, New York, NY, USA, 2007. ACM.

[RBH+07] Ralf Reussner, Steffen Becker, Jens Happe, Heiko Koziolek, Klaus Krogmann, and Michael Kuperberg. The Palladio Component Model. Technical Report 2007-21, Universität Karlsruhe (TH), 2007. http://digbib.ubka.uni-karlsruhe.de/volltexte/1000007341, last retrieved 2008-01-13.

[Rea] Real-Time and Distributed Systems Group, Carleton University. Layered Queueing Network Documentation. http://www.sce.carleton.ca/rads/lqns/lqn-documentation/. last retrieved 2008-01-13.

[Reu01a] R. Reussner. Enhanced component interfaces to support dynamic adaption and extension. In *HICSS '01: Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)-Volume 9*, page 9043, Washington, DC, USA, 2001. IEEE Computer Society.

[Reu01b] Ralf H. Reussner. *Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten*. Dissertation, Universität Karlsruhe (TH), July 2001.

[RFB04] Ralf H. Reussner, Viktoria Firus, and Steffen Becker. Parametric Performance Contracts for Software Components and their Compositionality. In Wolfgang Weck, Jan Bosch, and Clemens Szyperski, editors, *Proceedings of the 9th International Workshop on Component-Oriented Programming (WCOP 04)*, June 2004.

[RL80] M. Reiser and S. S. Lavenberg. Mean-value analysis of closed multichain queuing networks. *J. ACM*, 27(2):313–322, 1980.

[RPS03] Ralf H. Reussner, Iman H. Poernomo, and Heinz W. Schmidt. Reasoning on software architectures with contractually specified components. In A. Cechich, M. Piattini, and A. Vallecillo, editors, *Component-Based Software Quality: Methods and Techniques*, number 2693 in LNCS, pages 287–325. Springer, 2003.

[RRMP08] Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and Frantisek Plasil, editors. *The Common Component Modeling Example: Comparing Software Component Models*, volume to appear of *LNCS*. Springer, Heidelberg, 2008.

[RS95] J. A. Rolia and K. C. Sevcik. The method of layers. *IEEE Trans. Softw. Eng.*, 21(8):689–700, 1995.

[RS02] Ralf H. Reussner and Heinz W. Schmidt. Using parameterised contracts to predict properties of component based software architectures. In Ivica Crnkovic, Stig Larsson, and Judith Stafford, editors, *Proc. 9th IEEE Conference and Workshops on Engineering of Computer-Based Systems (ECBS'02)*, 4 2002.

[RSP03] Ralf H. Reussner, Heinz W. Schmidt, and Iman H. Poernomo. Reliability prediction for component-based software architectures. *J. Syst. Softw.*, 66(3):241–252, 2003.

[RZ07] Simone Röttger and Steffen Zschaler. Tool support for refinement of non-functional specifications. *Journal on Software and Systems Modelling (SoSyM)*, 6(2), June 2007.

[SGM02] Clemens Szyperski, Daniel Gruntz, and Stephan Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2002.

[Sim06] Erik Simmons. The usage model: Describing product usage during design and development. *IEEE Software*, 23(3):34–41, May/June 2006.

[SKK+01] Murali Sitaraman, Greg Kuczycki, Joan Krone, William F. Ogden, and A.L.N. Reddy. Performance specification of software components. In *Proc. of SSR '01*, 2001.

[SLC+05] Connie U. Smith, Catalina M. Llado, Vittorio Cortellessa, Antinisca Di Marco, and Lloyd G. Williams. From UML models to software performance results: an SPE process based on XML interchange formats. In *Proc. 5th international workshop on Software and performance (WOSP'05)*, pages 87–98, New York, NY, USA, 2005. ACM Press.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 263–272, New York, NY, USA, 2005. ACM Press.

[SMF+07] Jayshankar Sankarasetty, Kevin Mobley, Libby Foster, Tad Hammer, and Terri Calderone. Software Performance in the Real World: Personal Lessons from the Performance Trauma Team. In *Proc. 6th International Workshop on Software and Performance (WOSP'07)*, pages 201–208, New York, NY, USA, 2007. ACM.

[Smi90] C.U. Smith. *Performance Engineering of Software Systems*. Addision-Wesley, 1990.

[Smi01] Connie U. Smith. Origins of software performance engineering: Highlights and outstanding problems. In *Performance Engineering*, volume 2047 of *LNCS*, pages 96–118. Springer, 2001.

[Smi02] Connie U. Smith. *Performance Solutions: A Practical Guide To Creating Responsive, Scalable Software*. Addison-Wesley, 2002.

[SPJN98] Christiane Shousha, Dorina Petriu, Anant Jalnapurkar, and Kennedy Ngo. Applying performance modelling to a telecommunication system. In *Proc. 1st international workshop on Software and performance (WOSP '98)*, pages 1–6, New York, NY, USA, 1998. ACM.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer Verlag, Wien, 1973.

[Sun] Sun Microsystems. Java EE at a Glance. `http://java.sun.com/javaee/`. last retrieved 2008-01-13.

[SVC06] T. Stahl, M. Voelter, and K. Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[SW97] Fahim Sheikh and Murray Woodside. Layered Analytic Performance Modelling of a Distributed Database System. In *Proc. 17th International Conference on Distributed Computing Systems (ICDCS '97)*, page 482, Washington, DC, USA, 1997. IEEE Computer Society.

[Tip94] Frank Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, 1994.

[Tri01] Kishor Trivedi. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. Wiley & Sons, 2nd edition, 2001.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.

[VDTD07] T. Verdickt, B. Dhoedt, F. De Turck, and P. Demeester. Hybrid Performance Modeling Approach for Network Intensive Distributed Software. In *Proc. 6th International Workshop on Software and Performance (WOSP'07)*, ACM Sigsoft Notes, pages 189–200, February 2007.

[VHBP00] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *ASE '00: Proceedings of the 15th IEEE international conference on Automated software engineering*, page 3, Washington, DC, USA, 2000. IEEE Computer Society.

[Voa98] Jeffrey M. Voas. Certifying off-the-shelf software components. *Computer*, 31(6):53–59, 1998.

[Voa99] Jeffrey M. Voas. Certifying software for high-assurance environments. *IEEE Softw.*, 16(4):48–54, 1999.

[Voa00] Jeffrey M. Voas. Will the real operational profile please stand up? *IEEE Softw.*, 17(2):87–89, 2000.

[Wei81] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[Wey98] Elaine J. Weyuker. Testing component-based software: A cautionary tale. *IEEE Softw.*, 15(5):54–59, 1998.

[WFP07] Murray Woodside, Greg Franks, and Dorina Petriu. The Future of Software Performance Engineering. In *Future of Software Engineering (FOSE '07)*, pages 171–187, Los Alamitos, CA, USA, May 2007. IEEE Computer Society.

[Whi92] James A. Whittaker. *Markov chain techniques for software testing and reliability analysis*. PhD thesis, University of Tennessee: Knoxville, TN, 1992.

[WMW03] Xiuping Wu, David McMullan, and Murray Woodside. Component-based Performance Prediction. In *Proc. 6th ICSE Workshop on Component-based Software Engineering (CBSE'06)*, pages 13–18, 2003.

[WNHM86] C. M. Woodside, E. Neron, E. D. Hos, and B. Mondoux. An active-server model for the performance of parallel programs written using rendezvous. *Journal of Systems and Software*, 6(1-2):125–132, 1986.

[WNPM95] C. Murray Woodside, John E. Neilson, Dorina C. Petriu, and Shikharesh Majumdar. The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. *IEEE Trans. Comput.*, 44(1):20–34, 1995.

[Woi94] Denise Woit. *Operational Profile Specification, Test Case Generation, and Reliability Estimation for Modules*. PhD thesis, Queen's University, Kingston, Ontario, Canada, 1994.

[Woo84] C. M. Woodside. An active-server model for the performance of parallel programs written using rendezvous. In *Proc. IFIP Workshop on Performance Evaluation of Parallel Systems*, Grenoble, Dec. 13 - 15 1984.

[Woo89] C. Murray Woodside. Throughput calculation for basic stochastic rendezvous networks. *Perform. Eval.*, 9(2):143–160, 1989.

[Woo02] Murray Woodside. Tutorial Introduction to Layered Modeling of Software Performance. `http://www.sce.carleton.ca/rads/lqns/lqn-documentation/tutorialg.pdf`, May 2002. last retrieved 2008-01-13.

[WP93] James A. Whittaker and J. H. Poore. Markov analysis of software specifications. *ACM Trans. Softw. Eng. Methodol.*, 2(1):93–106, 1993.

[WPS02] Murray Woodside, Dorin Petriu, and Khalid Siddiqui. Performance-related completions for software specifications. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 22–32, New York, NY, USA, 2002. ACM Press.

[WR94] Claes Wohlin and Per Runeson. Certification of software components. *IEEE Trans. Softw. Eng.*, 20(6):494–499, 1994.

[WRH+00] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering: an Introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[WS03] Lloyd G. Williams and Connie U. Smith. Making the business case for software performance engineering. In *Proceedings of CMG*, 2003. last retrieved 2008-01-13.

[WT94] James A. Whittaker and Michael G. Thomason. A markov chain model for statistical software testing. *IEEE Trans. Softw. Eng.*, 20(10):812–824, 1994.

[Wu03] Xiuping Wu. An approach to predicting performance for component based systems. Master's thesis, Carleton University, Ottawa, Canada, July 2003.

[WV00] James A. Whittaker and Jeffrey Voas. Toward a more reliable theory of software reliability. *Computer*, 33(12):36–42, 2000.

[WVCB01] C. Murray Woodside, Vidar Vetland, Marc Courtois, and Stefan Bayarov. Resource function capture for performance aspects of software components and sub-systems. In *Performance Engineering, State of the Art and Current Trends*, pages 239–256, London, UK, 2001. Springer-Verlag.

[WW04] Xiuping Wu and Murray Woodside. Performance Modeling from Software Components. In *Proc. 4th International Workshop on Software and Performance (WOSP'04)*, volume 29, pages 290–301, New York, NY, USA, 2004. ACM Press.

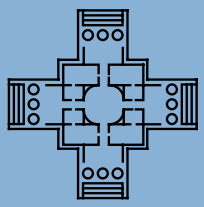
[WWL04] Pengfei Wu, Murray Woodside, and Chung-Horng Lung. Compositional layered performance modeling of peer-to-peer routing software. In *Proc. of the IEEE International Conference on Performance, Computing, and Communications*, pages 231– 238, 2004.

[XOWM06] Jing Xu, Alexandre Oufimtsev, Murray Woodside, and Liam Murphy. Performance modeling and prediction of enterprise javabeans with layered queuing network templates. *SIGSOFT Softw. Eng. Notes*, 31(2):5, 2006.

[Yac02] Sherif M. Yacoub. Performance analysis of component-based applications. In *SPLC 2: Proceedings of the Second International Conference on Software Product Lines*, pages 299–315, London, UK, 2002. Springer-Verlag.

[Zsc04] Steffen Zschaler. Formal specification of non-functional properties of component-based software. In Jean-Michel Bruel, Geri Georg, Heinrich Hussmann, Ileana Ober, Christoph Pohl, Jon Whittle, and Steffen Zschaler, editors, *Workshop on Models for Non-functional Aspects of Component-Based Software (NfC'04) at UML conference 2004*, September 2004. Technical Report TUD-FI04-12 Sept.2004 at Technische Universität Dresden.

[Zsc07] Steffen Zschaler. *A Semantic Framework for Non-functional Specifications of Component-Based Systems*. Dissertation, Technische Universität Dresden, Dresden, Germany, April 2007.



The Karlsruhe Series on Software Design and Quality

Edited by Prof. Dr. Ralf Reussner

This series reports on research in Karlsruhe for the engineering foundations of software design.

Despite the increasing computational power of modern computers, many large, distributed software systems still suffer from performance problems today. To avoid design-related performance problems, modeldriven performance prediction methods analyse the response times, throughputs, and resource utilisations of systems under development based on design documents before and during implementation. For component-based software systems, existing prediction methods neglect the performance influence of different usage profiles (i.e., the number of requests and the included parameter values) in their specification languages, which limits their prediction accuracy. This thesis proposes new modelling languages and according model transformations, which allow a reusable description of usage profile dependencies in component-based software systems. The thesis includes an experimental evaluation, which shows that predictions based on the newly introduced models can support design decisions for scenarios, whose performance is influences by different usage profiles.

ISSN: 1867-0067
ISBN: 978-3-86644-272-6

www.uvka.de