# A Model Transformation from the Palladio Component Model to Layered Queueing Networks

Heiko Koziolek[1], Ralf Reussner[2]

[1]Graduate School Trustsoft *
University of Oldenburg, Germany and
[2]Chair for Software Design and Quality
University of Karlsruhe, Germany
{koziolek,reussner}@ipd.uka.de

**Abstract.** For component-based performance engineering, software component developers individually create performance specifications of their components. Software architects compose these specifications to architectural models. This enables assessing the possible fulfilment of performance requirements without the need to purchase and deploy the component implementations. Many existing performance models do not support component-based performance engineering but offer efficient solvers. On the other hand, component-based performance engineering approaches often lack tool support. We present a model transformation combining the advanced component concepts of the Palladio Component Model (PCM) with the efficient performance solvers of Layered Queueing Networks (LQN). Joining the tool-set for PCM specifications with the tool-set for LQN solution is an important step to carry component-based performance engineering into industrial practice. We validate the correctness of the transformation by mapping the PCM model of a component-based architecture to an LQN and conduct performance predictions.

## 1 Introduction

Although the computational power of modern hardware is constantly increasing, many IT companies still face serious performance problems in their systems. This can lead to reduced user satisfaction and high maintenance costs [30].

The increasing complexity of modern software systems makes it hard to analyse performance properties at low abstraction levels. The idea of component-based software performance engineering (CBSPE) is to let software architects reason on the performance properties of their systems during design time at an architectural level using performance specifications provided by different component vendors. This enables them to manage the complexity of the performance model, to identify performance-critical components, and to avoid poor designs.

* This work is supported by the German Research Foundation (DFG), grants GRK 1076/1 and RE 1674/1-2

For component developers, it is not trivial to supply performance specifications of their components. As components shall be composed and deployed independently, component developers cannot make assumptions on how software architects compose components with others, how components will be deployed, and how users will execute them. All these factors influence the performance properties of a component. Therefore, component developers have to supply *parametrised* specifications, which software architects can adapt to different environments.

Researchers have proposed several approaches with parametrised specifications to support CBSPE (e.g., [5, 9, 6]). However, none of these approaches has reached industrial maturity due to still limited parametrisation concepts and due to a lack of tool support [16]. The Palladio Component Model (PCM) [4] is another proposal for CBSPE. It features component performance specification parametrised for different resource environments, usage profiles, and calls to required services. There is a discrete-event simulator for performance analysis of PCM instances, which, however, can be time-consuming for non-trivial systems.

Approaches for CBSPE can build on analytical methods for monolithic performance models, after the software architect has composed the individual component performance specifications, and tools have resolved their parametrisations. A mature monolithic performance model for distributed software systems with an efficient analytical solver is provided by Layered Queueing Networks (LQNs) [25]. Although there is an extension for LQNs to support CBSPE [32], its parametrisation concept is still limited. Therefore, we do not use this extension in this work.

In this paper, we introduce a fully automated model transformation from PCM to LQN. Software architects can use this transformation and the connected LQN solver to assess the performance of a PCM instance. With the PCM as input model, they can easily change parameter values in the PCM instance and analyse different settings. Because the LQN solver relies on Mean-Value-Analysis (MVA) and carries out an approximative performance prediction, it allows quicker performance analysis than running the PCM discrete-event simulator in many cases.

The contributions of the paper are (i) a model transformation from PCM to LQN, and (ii) a case study, where the transformation helped to analyse the performance of a component-based system. A part of the transformation (i.e., solving parameter dependencies) can be reused for other model transformations. To the best of our knowledge the transformation in this paper is the first implemented and validated transformation from a component-based modelling language to LQNs.

The remainder of this paper is organised as follows: Section 2 surveys related work in the area of component-based performance engineering and model transformations for LQN. Section 3 briefly introduces the basic concepts of PCM and LQN. Section 4 describes the two-step model transformation involving the solution of parameter dependencies and the mapping to LQNs. Section 5 presents a case study applying the transformation on the model of a component-based

system. Section 6 discusses limitations of the transformation, before Section 7 concludes the paper.

## 2 Related Work

The area of software performance engineering (SPE) originates from the pioneering work of Connie Smith [26]. Balsamo et al. [1] have surveyed several approaches for SPE, which use annotated, UML-like design models and transform them into performance models, such as queueing networks, stochastic process algebra, or stochastic Petri nets. Becker et al. [3] compare different approaches for CBSPE.

Several approaches introduce model transformations targeting LQNs. The source models are annotated UML diagrams [23, 8, 31, 12], Use Case Maps [22], and CSM [21]. These approaches do not support the specifics of component-based systems. Grassi et al. [11] have defined the intermediate modelling language KLAPER, which shall ease model transformations between different component-based design models and performance models. A KLAPER to LQN mapping is under development, and the work in this paper could be adapted to incorporate this mapping. However, the performance annotations in KLAPER so far do not follow defined semantics, which complicates automatic transformations.

Though some researchers have used LQNs to model component-based systems (e.g. [28, 29]), these approaches create single monolithic models, from which individual component specifications cannot be reused for different systems, because they lack the necessary parametrisation.

Wu et al. [32] have extended LQNs with the Component-based Modeling Language (CBML), which adds explicit provided and required interfaces to parts of LQNs and therefore enables replacing these parts with other LQN parts conforming to the same interfaces. This extension also features a form of parametrisation, which for example allows adapting the number of thread instances available to a component. The parametrisation however does not refer to input or output parameters of a component service, which is supported by the PCM.

Besides LQNs, other performance models have been used to analyse the performance of component-based software systems. Liu et al. [18] focus on EJB-based systems and have created a benchmark for application servers. Combining the benchmark results with an application model yields a queueing network, which allows analysing an application architecture for different workloads. The performance models created by this approach rely on certain EJB patterns and are hardly reusable in different settings. Kounev [15] uses Queueing Petri Nets (QPN) to model the SPEC jAppServer 2004, which consists of several software components. However, the resulting model is monolithic and cannot be decomposed into individual, reusable models for single components.

# 3 Foundations

## 3.1 Palladio Component Model

The Palladio Component Model (PCM) is a meta-model for the specification of component-based software systems and especially targets performance predictions [4]. Besides the specification of software components (according to Szyperski's definition [27]) and connectors, it additionally allows modelling hardware resources and resource demands of components. While UML models annotated with the UML SPT profile [19] could be used to model similar information as in the PCM, the PCM includes more advanced component concepts than the UML and features a parametrisation concept, which enables independent modelling by different component developers. The PCM is divided into sev-

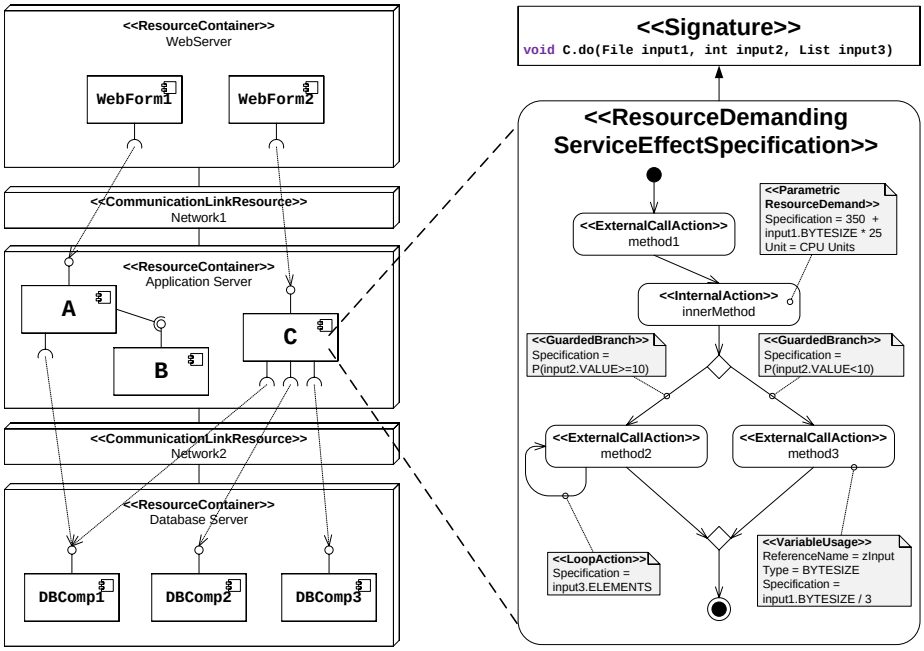



**Fig. 1.** A simple example PCM Instance

eral sub-models targeting specific developer roles. Component developers specify behavioural abstractions of their components and put them into repositories. Software architects retrieve these specifications during design time and compose them to the model of a complete software system. System deployers provide a model that specifies the hardware environment and the allocation of components to resources. Finally, domain experts use the PCM to specify the usage of the system in terms of number of users, user flow, and input parameters.

As the PCM contains more than 100 meta-classes, we only provide a simple example for a PCM instance here (Fig. 1) in a UML-like concrete syntax to

give the reader an idea of the PCM's modelling capabilities (more details in [4]). The figure's left hand side contains an example component-based software architecture (provided by a software architect) and its allocation to hardware resources (provided by a system deployer).

Each component may include an abstract behavioural description for each of its provided services (specified by the component developer), which is called Resource Demanding Service Effect Specification (RDSEFF). It specifies the resource demands of the service and its calls to required services. Fig. 1 depicts an RDSEFF for the service `do` of the component `C` on the right hand side. The service first calls an external service `method1` and then uses the CPU of the application server (`internalMethod`). The component developer specifies the resource demand in an abstract unit ("CPU-Units"), which can be converted into a timing values once the system deployer has specified the execution time for a CPU unit. A single internal action can represent a large amount of code in a single model element, thereby creating an abstraction from the implementation.

In this case, the resource demand is specified including a dependency to the size of the service's input parameter `input1`. Once the domain expert specifies the size of this input parameter for the given application context (e.g., 1000), the actual resource demand can be resolved (e.g., 25350 CPU Units). Because of the parametrisation, the specification can be easily adapted for different usages and hardware environments (not shown here) if the component is reused. Besides parametrised resource demands, RDSEFFs also allow parametrised branch transitions, loop iteration numbers, and input parameters to required services as shown in Fig. 1. The RDSEFF parametrisation allows modelling performance annotations in dependency to the data flow between components, whereas in other approaches (e.g., LQNs) the parametrisation only refers to single components. There are several extensions for the PCM (e.g., [13, 2]) to reflect performance-relevant influences by the middleware.

The PCM is specified in Ecore from the Eclipse Modelling Framework. There are several graphical editors for the specification of PCM instances. There is also a discrete-event simulation for PCM instances called SimuCom [4], which enables deriving performance metrics such as response times, throughputs, and resource demands of a complete system model, but can be time-consuming for large models, because it supports arbitrary distributed service times. Finally, several reverse engineering tools are under development [7, 14], which shall semi-automatically derive components and RDSEFFs given arbitrary Java code.

### 3.2 Layered Queueing Networks

Layered Queueing Networks (LQN) [10] are a performance model in the class of extended queueing networks. Other than plain QNs, LQNs model software entities and their communication explicitly in a hierarchical structure. Like the PCM, LQNs target the performance analysis of distributed business informations systems, but unlike the PCM they do not support independent specification of individual software components. There is an approximative, analytical solver based on Mean-Value Analysis (MVA) for LQNs including M/M/n queues [25].

As an example, Fig. 2 shows a simple LQN instance in the standard concrete syntax. It is an acyclic graph and consists of *processors* (circles) and *tasks* (parallelograms). Processors model hardware entities such as CPUs, hard disks, or networks. Tasks model software entities, such as components, application servers, databases, semaphores, or buffers. Tasks are arranged in a layered hierarchy, where tasks from upper layers may send requests to tasks from lower layers. Both processors and tasks contain a request queue (not depicted in the figure), from which they serve waiting requests according to a specific scheduling discipline (e.g., FCFS or Processor Sharing).

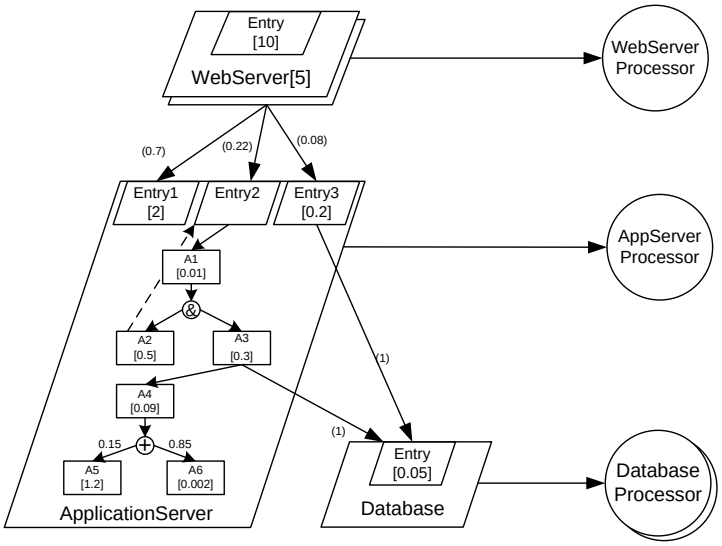



**Fig. 2.** A simple example LQN Instance

Each task can contain multiple *entries*, which model the services provided by the software entity. Entries either directly specify a resource demand to the underlying processor of the tasks, or include a control flow graph containing multiple *activities*, which issue such demands. Both entries and activities can also make calls to the entries of tasks on lowers layers of the LQNs. These calls can be synchronous (i.e., blocking the caller) or asynchronous (i.e., the control flow of the caller continues immediately after issuing the request).

The control flow graphs for activities support sequences, branches, loops, and forks. Other than in the PCM, branch probabilities and loop iteration numbers have to be specified as constant values and cannot depend on input parameters. Resource demands by activities or entries specify execution times as mean values of exponential distributions.

If an entry does not include a control flow graph, its execution may consists of up to three so-called *phases*, where each phase can request processing from

the underlying processor or call other entries. The implicit semantics of the first phase is that the caller of the entry containing the phase blocks until it is finished. The entry then generates a reply for the caller, after which the caller continues execution. Concurrently, the entry executes the second and third phase asynchronously from the caller. This models a common communication pattern in distributed systems, which tries to ensure a high responsiveness by returning control to clients as early as possible.

The top-most tasks in an LQN are called reference tasks. They model clients and may include open or closed workloads. Open workloads specify an arrival rate for incoming requests and do not bound the number of requests issued to the system. Closed workloads specify a bounded number of users circulating in the system (the user population). After completing execution of all requests, a user re-enters the system after a given think time.

## 4 Model Transformation

### 4.1 Process

The model transformation and solution process from PCM instances to LQN instances contains multiple steps (Fig. 3). First the different developer roles specify their parts of the PCM instance. After the domain expert has created the usage model, the PCM instance is complete and can be checked automatically for syntactical inconsistencies.

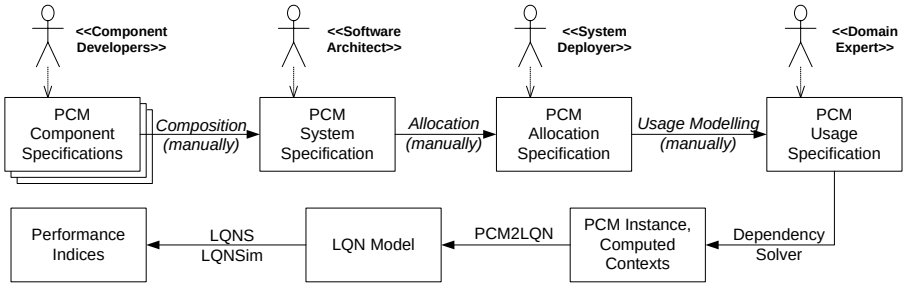



**Fig. 3.** Modelling and Transformation Process

The Dependency Solver (DS) takes the complete PCM model as input and propagates parameter values specified in the usage model through all RDSEFFs, substituting parameter references in these specifications with the actual values (Section 4.2). This step creates resource demands, branch probabilities, and loop iteration numbers without parameter dependencies. Afterwards, the tool PCM2LQN is responsible for mapping the model to an LQN instance (Section 4.3) and executing the LQN solver for the performance prediction. The tool-chain is fully automated after starting the DS and embedded into the PCM bench.

### 4.2 Transformation 1: Dependency Solver

The DS combines the sub-models from the different developer roles and removes the parametrisation from RDSEFF instances, so that they are prepared for a mapping to a performance model. To clarify this process, we will first briefly describe the PCM context model.

The PCM strictly separates information about the context (i.e., the composition, allocation, and usage) of a component from its own behavioural specification, because this information is unknown to the component developer. Software architects create a so-called *assembly context* for each component instance they compose into an architecture. It stores the component instance's binding to other components. There can be multiple assembly contexts for a single component type in an architecture, as a software architect can use multiple instances of the same component in the same architecture.

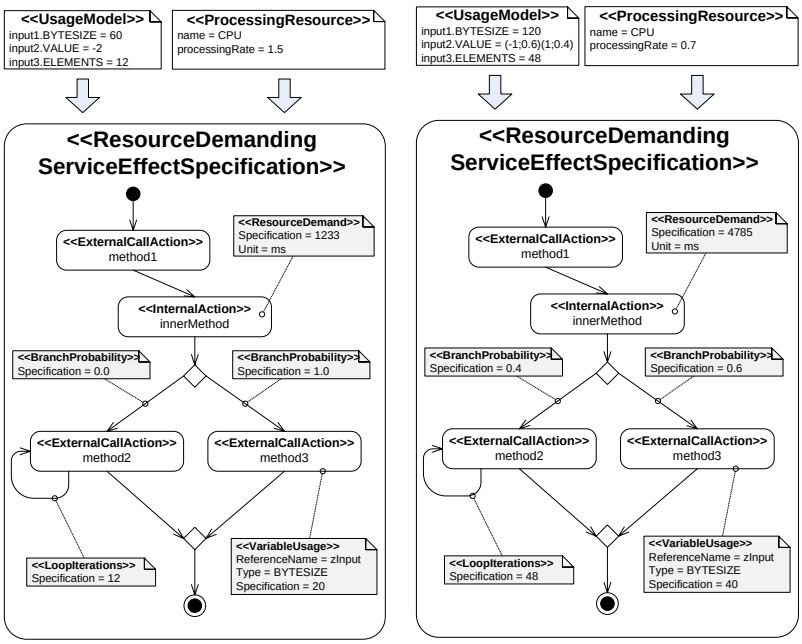
System deployers create a so-called *allocation context* for each *assembly context* specifying the component instance's deployment to a particular hardware resource. The usage of a component (i.e., the number of invocations and the used parameter values) only needs to be specified at the system boundaries for components directly interacting with users. The domain expert creates a so-called usage model, which stores this information. The DS then traverses all RDSEFFs using the binding specification from the assembly contexts and propagates the parameter values from the usage model through the architecture.

Consider the example in Fig. 4. It depicts the output of the Dependency Solver after processing the RDSEFF from Fig. 1 in two different contexts using the usage model and processing resource specification at the top of the figure. For example, the `ResourceDemand` of the left-hand side RDSEFF results from the `ParametricResourceDemand` ($350 +$ `input1.BYTESIZE` $* 25$) seen before in Fig. 1. The DS has substituted the actual parameter value specified in above's usage model ($350 + 60 * 25 = 1850$) and divided the expression by the processing rate from the processing resource ($1850/1.5 = 1233$).

The PCM allows component developers to specify parameter dependencies referring to the value, bytesize, length (for collections), or other performance-relevant properties of a parameter [16]. The dependencies may include arithmetic expressions (+,-,*,/) on resource demands or loop iteration numbers, and boolean expressions ($=, <, >, \leq, \geq, AND, OR$) on branching guards (cf. Fig. 1).

The PCM does not only support characterising parameter values with constant values, but also probability distributions. For example, the domain expert could specify `a.BYTESIZE = IntPMF[(10;0.2) (20;0.3) (30;0.5)]` in the usage model, meaning that the size of `a` in bytes is 10 with a probability of 0.2. Then, solving the parameter dependency for the example (`2 * a.BYTESIZE`) by the DS would result in a `RandomVariable` with the value: `IntPMF[(20;0.2) (30;0.3) (40;0.5)]`.

Notice that parameter dependencies need not exactly reflect the precise, actual dependencies given by the code of the component, which is for example often impractical for large components. A coarse abstraction of the dependency focussing on the performance impact of a parameter is often sufficient.




(a) RDSEFF in Context 1 (b) RDSEFF in Context 2

**Fig. 4.** Output of the Dependency Solver

The DS stores all solved expressions for parameter dependencies in the so-called "computed context model", which is a decorator model for the PCM. It includes a computed usage context model (meta-model in Fig. 5) , which stores solved expressions for branch probabilities, loop iteration numbers and input/output parameter values. Furthermore it includes a computed allocation model (meta-model in Fig. 6), which stores solved expressions for resource demands. These models are separated, because they result from different information sources (i.e., the domain expert and the system deployer).

The model traversal by the DS starts with RDSEFFs of components at the system boundaries. If these RDSEFFs contains calls to other RDSEFFs, the DS successively also traverses those RDSEFFs. Upon finishing the traversal of an RDSEFFs, the DS returns to the calling RDSEFF and creates the `ExternalCallOutput` specification of the `ComputedUsageContext`, which may include a solved parameter dependency to the return value or output parameter characterisations specified in the called RDSEFF. The DS traverses each loop body in the RDSEFF only once, which is sufficient for solving the parameter dependencies.

After the DS has traversed the whole model and created all computed context models, this decorated PCM instance is ready for the transformation into a performance model. A more detailed description of the DS can be found in [16]. Although the following only describes the mapping to LQNs, transformations to

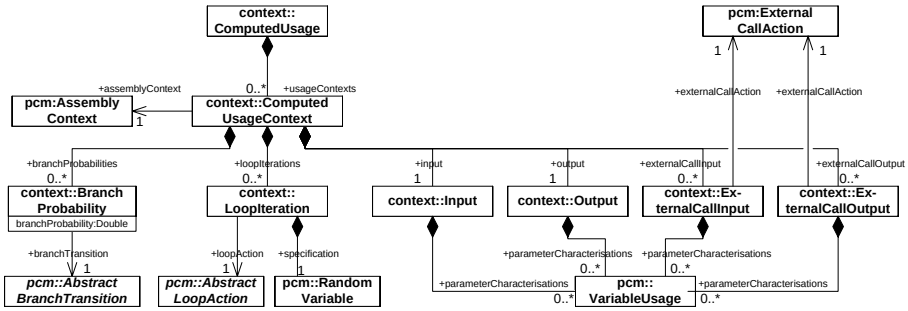



**Fig. 5.** Computed Usage Context (Meta-Model)

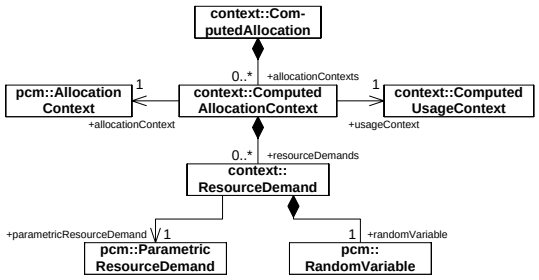



**Fig. 6.** Computed Allocation Context (Meta-Model)

other performance models can be applied at this point. For example, there is a transformation to Stochastic Regular Expressions [17].

### 4.3 Transformation 2: PCM2LQN

The second transformation PCM2LQN maps a PCM instance decorated with computed contexts to an LQN instance. This transformation is documented in detail in [16]. Due to space reasons, this papers describes the mapping with an example, provides an overview of the complete mapping, and highlights challenges of the transformation due to semantic gaps between PCM and LQN.

**Example** Fig. 7 demonstrates how PCM2LQN maps the RDSEFF from Fig. 4(b) into an LQN. Each RDSEFF is mapped into an LQN task with a task activity graph. Although PCM2LQN could map all RDSEFFs of a single component to a single LQN task with multiple entry activity graphs, this has not been implemented, as the LQN solvers so far do not support entry activity graphs.

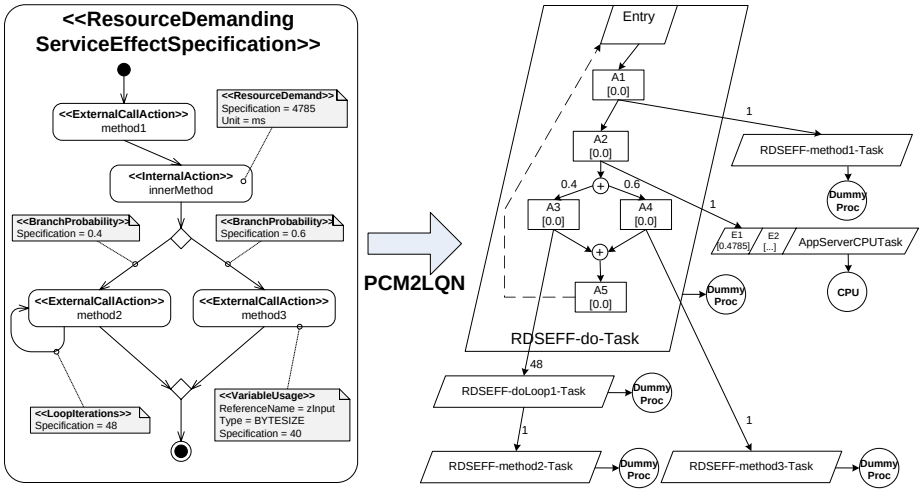



**Fig. 7.** PCM2LQN Example: Transforming an RDSEFF to LQN fragments

PCM2LQN transforms each `ExternalCallAction` into an LQN activity with zero host demand and a synchronous call to the task representing the called RDSEFF. The activities A1 and A4 in the example have resulted from this mapping.

PCM `InternalActions` model computations by a component service, which execute on the resources the component is deployed on. Every `InternalAction` can contain several `ResourceDemands` directed at specific resources, such as a CPU or hard disk. PCM2LQN creates an activity for the `InternalAction` and for each `ResourceDemand` (A2 in the example) and connects them sequentially.

RDSEFFs can reference multiple resources, but LQN tasks can run only on a single LQN processor. Thus, PCM2LQN converts PCM `ResourceDemands` to LQN entries, which are added to the task running on the processor created for the resource referenced by the `ResourceDemand` (E1 in the example). The activities created for the resource demand call these entries synchronously.

For the host execution demand of these entries, PCM2LQN either directly uses the PCM resource demand specification if is a constant or computes its expected value if is a probability distribution. This step is necessary as LQNs only support mean value resource demands. It lowers the accuracy of the model as information about the distribution gets lost.

The control flow constructs of branch and sequence can directly be mapped to their counterparts in LQN task graphs. PCM2LQN accesses the computed usage context for a given RDSEFF to retrieve the branch probabilities (0.4 and 0.6 in the example) and uses them in the task graph.

Although LQN activity graphs support loops, these loops may only contain a sequence of activities, but not branches or nested loops. PCM `Loop` bodies instead allow arbitrary behaviour. Therefore PCM2LQN creates a new LQN task for each loop body. Within this task, the LQN can include arbitrary behaviour and model the PCM loop body.

The tasks created for the loop body is called as often as the specified number of loop iterations (48 in the example). If the number of loop iterations is specified with a probability distribution, PCM2LQN uses its expected value for the number of calls to the loop body task.

The tasks created for RDSEFFs and loop bodies run on dummy LQN processors, which they do not use. PCM2LQN creates these processors to make the model valid for the LQN solvers. Only the LQN processors created for PCM `ProcessingResources` are actually used by LQN tasks. Their mapping is straight forward, as PCM2LQN can directly map their processing rates to the speed-factor of LQN processors, and their scheduling policies to LQN scheduling policies.

**Mapping Overview** Tab. 4.3 depicts a complete overview of the transformation. The first column refers to meta-classes from the PCM. The second and third column refer to the corresponding meta-classes from the LQN meta-model. The second column contains the main classes of the mapping, and the third column contains additionally created classes to make the LQN instance syntactically correct or to model control flow precedence.

In addition to the meta-classes, the values contained in the brackets refer to attributes of these classes. The table only includes the attributes of a meta-class if PCM2LQN maps to another value than the default value (documented in [24]), otherwise the attribute is left out in the table for brevity. For example, for an `Activity` of a LQN the default `hostDemand` is zero, therefore all `Activities` without a `hostDemand` attribute in the table have an implicit `hostDemand` of zero.

**Table 1.** Transformation PCM2LQN

| PCM | LQN | LQN - supplemental |
|---|---|---|
| *ResourceEnvironment* | | |
| prs:ProcessingResourceSpecification | Processor (scheduling=prs.schedulingPolicy, speedFactor=prs.processingRate) | Task, Entry |
| *UsageModel* | | |
| cw:ClosedWorkload | Task (scheduling=ref, thinkTime=expectedValue(cw.thinkTime), multiplicity=cw.population) | Processor, Entry |
| ow:OpenWorkload | Task (scheduling=ref), Entry (openArrivalRate=1/expectedValue(ow.interArrivalTime)) | Processor |
| sb:ScenarioBehaviour | TaskActivityGraph | |
| elsc:EntryLevelSystemCall | Activity(synchCall) | Precedence (pre=elsc, post=elsc.successor) |
| d:Delay | Activity(thinkTime=expectedValue(d.userDelay)) | Precedence (pre=d, post=d.successor) |
| b:Branch | Activity, Precedence(pre=b, postOR=bt_1..n), Precedence (preOR=bt_1..n, post=b.successor) | |
| bt:BranchTransition | ActivityOr(prob=bt.branchProbability) | |
| l:Loop | Activity (synchCall, callsMean=expectedValue(l.iterations)) | Processor, Task, Entry, Precedence (pre=l, post=l.successor) |
| *RDSEFF* | | |
| rdb:ResourceDemandingBehaviour | TaskGraph | Processor, Task, Entry |
| st:StartAction | - | |
| sp:StopAction | ReplyActivity, ReplyEntry | |
| eca:ExternalCallAction | Activity(synchCall) | Precedence (pre=eca, post=eca.successor) |
| ba:BranchAction | Activity, Precedence(pre=ba, postOR=abt_1..n), Precedence (preOR=abt_1..n, post=ba.successor) | |
| abt:AbstractBranchTransition | ActivityOr(prob=computedUsageContext(abt).branchProbability) | |
| la:LoopAction | Activity (synchCall, callsMean= expectedValue(computedUsageContext(l). iterations)) | Processor, Task, Entry, Precedence (pre=la, post=la.successor) |
| cia:CollectionIteratorAction | Activity (synchCall, callsMean= expectedValue(computedUsageContext(cia). iterations)) | Processor, Task, Entry, Precedence (pre=cia, post=cia.successor) |
| ia:InternalAction | Activity(hostDemand=0) | Precedence (pre=ia, post='first prd'), Precedence (pre='last prd', post=ia.successor) |
| prd:ParametricResourceDemand | Activity(synchCall), Entry, PhaseActivity(hostDemand=expectedValue(computedUsageContext(prd).resourceDemand)) | Precedence (pre=prd, post='next prd') |
| sva:SetVariableAction | - | |
| fa:ForkAction, sp:SynchronisationPoint | Activity, Precedence(pre=fa, postAND=rdb_1..n), Precedence (preAND=rdb_1..n, post=fa.successor) | |
| pr:PassiveResource | Task(schedDisc=semaphore), Entry (signal), Entry (wait) | |
| aa:AcquireAction | Activity(synchCall, dest='wait') | Precedence (pre=aa, post=aa.successor) |
| ra:ReleaseAction | Activity(synchCall, dest='signal') | Precedence (pre=ra, post=ra.successor) |

Several LQN classes reference each other using strings, which refer to the `name` attribute of other classes. The LQN's `Precedence` classes use this mechanism to connect individual `Activities` to an activity graph. The table does not include all reference strings used in the transformation as they add little value to understanding the transformation.

The mapping for PCM usage models is similar to mapping of RDSEFFs. With them, domain experts specify user behaviour in terms of workload, scenarios, and calls to RDSEFFs. PCM2LQN maps the included `ClosedWorkloads` to LQN reference tasks (i.e., scheduling=ref). Such tasks only emit requests, and cannot serve requests themselves. The attribute `population` (i.e., the number of concurrent users) of the `ClosedWorkload` is equivalent to the multiplicity attribute of the new reference task. The attribute `thinkTime` (i.e., the time a user waits before re-entering a scenario after completing it) is mapped to the LQN task think time.

PCM `OpenWorkloads` are also mapped to reference tasks. However, in this case their think time is 0.0 and their multiplicity is 1 (i.e., the default values). PCM2LQN transforms the `OpenWorkload's interArrivalTime` into a rate using the expected value of the specified probability distribution. This rate is used as the open arrival rate for the entry in the newly created reference task. PCM2LQN maps the rest of the usage model similarly to RDSEFFs, therefore we omit a detailed description.

In addition to the mappings shown in the example PCM2LQN also supports mapping RDSEFF `ForkActions`. They model the invocation of threads and their concurrent execution. The mapping to LQNs is similar to the mapping for branches. PCM2LQN uses an AND precedence to create the fork and creates new tasks for the forked behaviours. After they have finished execution, another precedence merges the forked control flow together again. So far, the mapping only supports synchronous forks.

In the PCM, components can have `PassiveResources`, which can be used to model semaphores or thread pools. LQNs use special tasks to model semaphores. These tasks have the scheduling discipline 'semaphore' and contain two entries named 'wait' and 'signal'. The first entry allows requesting the semaphore, while the second entry models returning the semaphore. PCM2LQN creates such a task for each `PassiveResource` in the PCM instance. The `AcquireAction` and `ReleaseActions` are mapped to activities with synchronous calls to the 'wait' entry or 'signal' entry respectively.

**Prototypical Implementation** PCM2LQN uses three visitors (implemented in Java) to traverse the PCM's `ResourceEnvironment`, `UsageModel`, and `RDSEFF` models. The navigation between the RDSEFFs is managed by using the assembly contexts and looking up the connected components in the PCM `System` specification.

PCM2LQN creates instances of an LQN meta-model in Ecore. This meta-model has been generated with EMF from the LQN-XML schema provided with the LQN tools (Version 3.12, cf. [24]). Once the visitors of PCM2LQN have

traversed the whole PCM instance, an object representation of the LQN instance has been created. Using the XML serialisation of EMF, PCM2LQN then saves this representation to an XML file, which is the input of the LQN solvers.

## 5 Case Study

The following case study serves to demonstrate the correctness of the model transformation introduced in this paper. We have modelled a component-based software system as a PCM instance and used the Dependency Solver described in Section 4.2 as well as PCM2LQN described in Section 4.3 to generate an LQN and run the LQN solvers for performance analysis. Additionally, the case study points out the benefits of a parametrised, component-based performance specification as the PCM, which enables model reuse and analysis of the impact of different usage profiles, hardware resources, and component compositions to performance.

The case study investigates the so-called "Business Reporting System" (BRS), which is loosely based on an industrial system. We only present performance predictions based on the model and do not provide comparisons with measured data. The validity of LQN performance predictions have been shown in former studies (e.g., [10]) and are out of scope for this paper. We assume that the PCM instance of the BRS with its control flow and resource demands reflects the performance properties of the modelled system well.

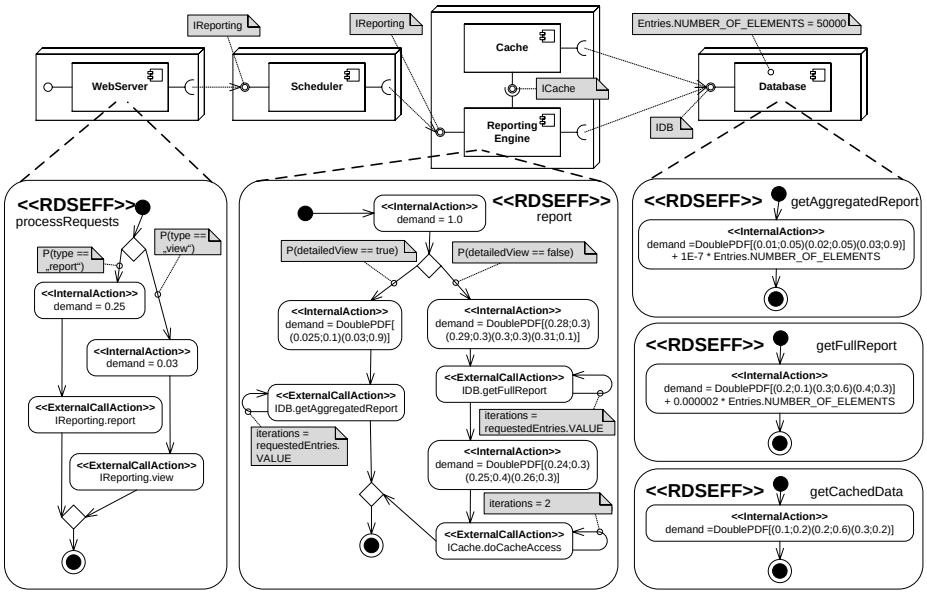



**Fig. 8.** Business Reporting System (Palladio Component Model)

The BRS is a 4-tier, web-based system to monitor and manage business data. On a high abstraction level, it consists of 5 software components (Fig. 8 at the top). Clients either request business reports or specific entries from the database via the `WebServer`. A `Scheduler` connects the `WebServer` with an `ApplicationServer`. The latter contains a component `ReportingEngine`, which manages the creation of reports, and a component `Cache`, which buffers data from the database for quick access. Both, the `ReportingEngine` and the `Cache` query the component `Database`, which stores a configurable amount of entries in its tables.

Fig. 8 shows PCM RDSEFFs for services of the `WebServer`, `ReportingEngine`, and `Database` at the bottom. The whole model consists of nine RDSEFFs, some have been omitted for brevity[1]. The first RDSEFF `processRequests` includes parameter dependencies, which determine branch probabilities according to the probabilities specified in the PCM usage model for the type of requested services (i.e., report or view). It also contains some constant resource demands to the `WebServer's` CPU.

The second depicted RDSEFF (`report`) from the `ReportingEngine` chooses a branch depending on whether the users request detailed reports or not. Detailed reports result in longer calls to the database. As the BRS also allows users to specify the number of entries in the generated reports, the loops in this RDSEFF are iterated as many times as the number of requested entries. Finally, this RDSEFF contains resource demands specified as probability density functions (PDF).

The three RDSEFFs on the right hand side of the figure represent services from the database system and do not include calls to other components. The resource demands specified in the upper two RDSEFFs depend on the number of entries specified in the `Database`. A larger number of entries results in longer queries. The component developer of the `Database` has made this relationship explicit, so that different software architects can adjust the model to their anticipated number of entries in the database.

The full PCM instance of the BRS additionally includes an usage model and a resource environment model, which are not illustrated here. Network traffic is considered negligible in the model.

Running the formerly described model transformations on the PCM instance of the BRS yields the LQN, schematically depicted in Fig. 9. The illustration only shows entries including non-zero host demands to the underlying processors, and only shows processors which are actually used by task. The complete model contains a processor for each task to make it valid for the solvers. The illustration also does not display the task activity graphs generated for the RDSEFFs.

Notice, how the loops of the RDSEFFs result in additional tasks and how the resource demands of RDSEFFs result in LQN entries as described in Section 4.3. For example, PCM2LQN has mapped the four resource demands of the RDSEFF `process` (seen in Fig. 8) to the entries E1-E4 of the `AppServerTask` in Fig. 9.

[1] The full PCM instance of the BRS system as well as PCM2LQN are available for download at http://www.palladio-approach.net.

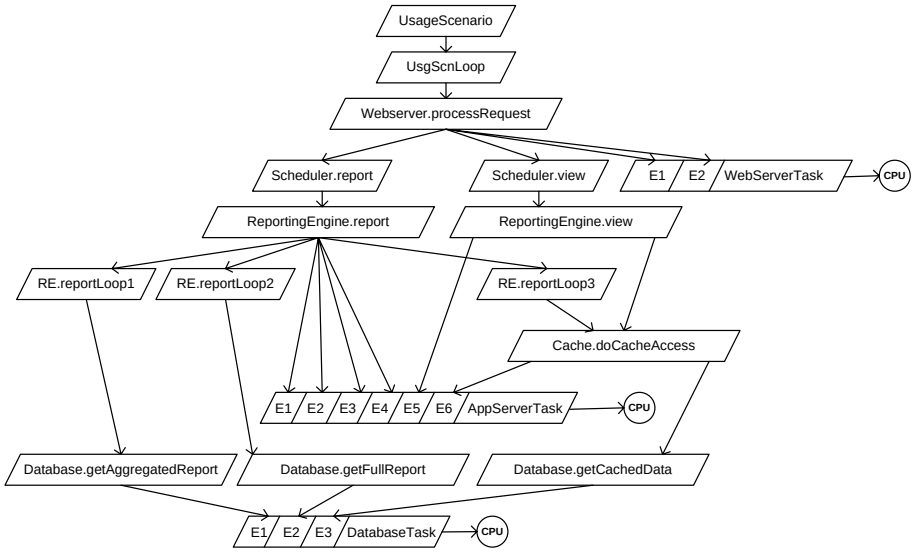



**Fig. 9.** Business Reporting System (Layered Queueing Network, Schematic Illustration)

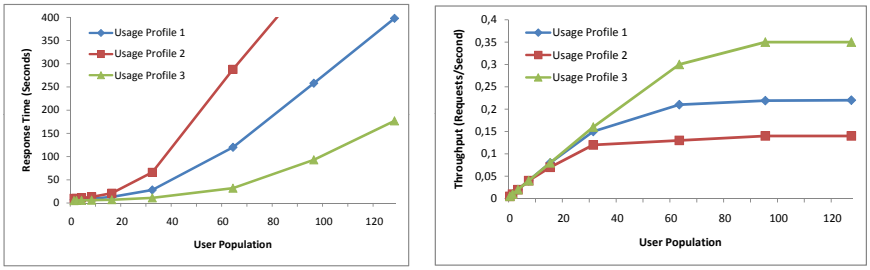
PCM2LQN determines the expected values for the probability density functions specified in the RDSEFFs and uses them in the LQN.

In our performance analysis, we predict the performance of the system for different usage profiles. This only requires changing the PCM usage model and not the PCM RDSEFFs, as the dependency solver automatically determines the branch probabilities, loop iterations numbers and resource demands for a given usage model. Here, we do not alter other possible parameters of the model, such as the speed of hardware resources or the composition of the components to keep the case study managable.

**Table 2.** Usage Profiles for the Business Reporting System

| | Usage Profile 1 | Usage Profile 2 | Usage Profile 3 |
|---|---|---|---|
| Type of request | 25% report,75% view | 40% report, 60% view | 10% report, 90% view |
| Number of requested entries | 10 | 5 | 7 |
| Detailed reports | 20% yes, 80% no | 70% yes, 30% no | 10% yes, 90% no |
| Entries in Database | 50000 | 10000 | 100000 |

Tab. 5 contains the three usage profiles used for the prediction. Users can change the type of requests, the number of entries per request, and decide whether they want detailed reports or not. Additionally, the number of entries in the database is part of the usage profiles and needs to be specified by the software architect. Here, the three usage profiles are not based on specific realistic settings, and only serve to demonstrate the prediction capabilities of the model.




(a) Response Times (b) Throughputs

**Fig. 10.** Performance Indices Business Reporting System

Running the transformation and the LQN solver for all usage profiles took less than 5 seconds in each case. We analysed the response time and throughput of the system for the different usage profile and an increasing user population (Fig.10). In some cases with a higher user population ($> 60$ users), the LQN solver did not converge, so that we used the LQN simulator to obtain the depicted results. The curves indicate that the system will be saturated for more than 64 users (usage profile 1), or more than 32 users (usage profile 2), or more than 96 users (usage profile 3).

## 6 Limitations

The model transformation introduced in this paper enables solving PCM instances with LQN solvers. It is beneficial for software architects, who can quickly analyse the performance properties of their design models. The parametrisation in PCM instances enables them to easily change the modelled usage profile, hardware environment, or component assembly and assess different design alternatives. However, there are still some open issues for the transformation:

- **Information Loss:** Some information within a PCM instance is lost when mapping to an LQN. For example, PCM2LQN computes the expected values of general distribution functions specified in an PCM instance and uses them in the LQN to specify resource demands and loop iteration numbers. Therefore, using LQN solvers for performance prediction is not useful if general distributions functions are of interest.
- **Exploiting more LQN constructs:** LQNs support more communication concepts between software entities than the PCM. For example, they allow asynchronous communication, forwarding of requests, and multiple phases. It is desirable to extend the PCM in the future to support more of these concepts, so that a larger number of systems can be analysed.
- **Incorporating Intermediate Modelling Languages:** KLAPER [11] and CSM [21] are intermediate modelling languages, which shall ease the implementation between design-oriented models, such as the PCM, and analysis-oriented models, such as LQNs. Transformations from these languages to

LQNs are planned, but not yet implemented. Once these transformations become available, the model transformation should be adapted to incorporate them.
- **Solver Feedback:** Mapping PCM instances to LQN instances and running the solver has been fully automated and integrated into the PCM bench. However, the current implementation simply prints the textual solver results to the screen, so that the performance analyst has to interpret them. For the future, a more sophisticated feedback of the solver results into the PCM instance would be desirable, so that LQNs become fully transparent for the analyst.
- **Standardised Transformation Language:** We have implemented the both the Dependency Solver and PCM2LQN as ad-hoc Java transformations. Once engines for standardised transformations languages such as QVT become available, it is desirable to use QVT to implement the transformation.
- **Standardised Design Model:** Instead of UML, the PCM is a proprietary modelling language specifically designed for the performance analysis of component-based software systems. So far, existing UML models cannot be reused without manual overhead when specifying a PCM instance. A transformation from UML models to PCM instances could enable reusing parts of existing UML models and lower the reservation of using the PCM in industry.

## 7 Conclusions

The model transformation introduced in this paper connects efficient performance solvers for monolithic software architectures to a component-based performance modelling language. The transformation bridges differences of LQNs and PCM instances, by for example mapping distribution functions to expected values and allowing components to access multiple resources. We have embedded the transformation into the PCM bench for modelling and analysing PCM instances, so that performance analysts can use the LQN solvers for quick performance predictions. While the solvers are more efficient than the current PCM simulator SimuCom, they only deliver mean-value performance indices instead of distribution functions.

Having component-based, parametrised performance specifications such as in the PCM has several benefits. It allows reusing the performance specification in different contexts such as hardware environments, usage profiles and component assemblies. PCM RDSEFFs shall be stored in public repositories, so that different software architects can incorporate them into their architectural models. The parametrisation allows the different participating developer roles to model independently from each other. As RDSEFFs specify resource demands, loop iteration numbers, and branch probabilities in dependency to parameter values, it is easily possible to adjust the specification for different usage profiles. This is usually not possible in monolithic models (e.g., annotated UML diagrams), where for example the dependency between a branch probability and input parameters is not explicitly specified.

For the future, we plan to implement the transformation in a standardised transformation language such as QVT [20]. It is also desirable to map PCM instances into intermediate modelling languages such as CSM [21] or KLAPER [11] to enable transformation into even more performance models and exploit the specifics of these models. Another area of improvement is the feedback of the solver results into the PCM model, so that the performance models become fully transparent for software architects.

## References


1. 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.
2. Steffen Becker. Coupled Model Transformations. In *Proc. 7th International Workshop on Software and Performance (WOSP'08)*. ACM Sigsoft, June 2008. To Appear.
3. 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.
4. 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.
5. 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.
6. Egor Bondarev, Peter de With, Michel Chaudron, and Johan Musken. Modelling of Input-Parameter Dependency for Performance Predictions of Component-Based Embedded Systems. In *Proc. of the 31th EUROMICRO Conference (EUROMICRO'05)*, 2005.
7. Landry Chouambe, Benjamin Klatt, and Klaus Krogmann. Reverse Engineering Software-Models of Component-Based Systems. In *Proc. of the 12th European Conference on Software Maintenance and Reengineering (CSMR'08)*, Athens, Greece, April1–4 2008. IEEE. To appear.
8. Andrea D'Ambrogio. A model transformation framework for the automated building of performance models from UML models. In *Proc. 5th International Workshop on Software and Performance (WOSP'05)*, pages 75–86, New York, NY, USA, 2005. ACM Press.
9. 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.
10. Greg Franks. *Performance Analysis of Distributed Server Systems*. PhD thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada, December 1999.

11. 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.
12. Gordon P. Gu and Dorina C. Petriu. From UML to LQN by XML algebra-based model transformations. In *Proc. 5th International workshop on Software and Performance (WOSP'05)*, pages 99–110, New York, NY, USA, 2005. ACM.
13. Jens Happe, Holger Friedrichs, Steffen Becker, and Ralf Reussner. A Configurable Performance Completion for Message-Oriented Middleware. In *Proc. 7th International Workshop on Software and Performance (WOSP'08)*. ACM Sigsoft, June 2008. To Appear.
14. 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.
15. 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.
16. Heiko Koziolek. *Parameter Dependencies for Reusable Performance Specifications of Software Components*. PhD thesis, University of Oldenburg, Germany, March 2008.
17. 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.
18. Yan Liu, Alan Fekete, and Ian Gorton. Design-level performance prediction of component-based applications. *IEEE Trans. Softw. Eng.*, 31(11):928–941, 2005.
19. Object Management Group (OMG). UML Profile for Schedulability, Performance and Time, 2005. last retrieved 2008-01-13.
20. Object Management Group (OMG). MOF QVT final adopted specification (ptc/05-11-01), 2006. last retrieved 2008-01-13.
21. 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.
22. 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.
23. 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.
24. Real-Time and Distributed Systems Group, Carleton University. Layered Queueing Network Documentation. last retrieved 2008-01-13.
25. J. A. Rolia and K. C. Sevcik. The method of layers. *IEEE Trans. Softw. Eng.*, 21(8):689–700, 1995.
26. C.U. Smith. *Performance Engineering of Software Systems*. Addision-Wesley, 1990.
27. Clemens Szyperski, Daniel Gruntz, and Stephan Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2002.

28. Alexander Ufimtsev and Liam Murphy. Performance modeling of a JavaEE component application using layered queuing networks: revised approach and a case study. In *Proc. International Workshop on Specification and Verification of Component-based Systems (SAVCBS '06)*, pages 11–18, New York, NY, USA, 2006. ACM.
29. 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.
30. 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.
31. Murray Woodside, Dorina C. Petriu, Dorin B. Petriu, Hui Shen, Toqeer Israr, and Jose Merseguer. Performance by unified model analysis (puma). In *WOSP '05: Proceedings of the 5th international workshop on Software and performance*, pages 1–12, New York, NY, USA, 2005. ACM Press.
32. 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.