# Predicting the Performance of Component-Based Software Architectures with different Usage Profiles

Heiko Koziolek, Steffen Becker, Jens Happe

Graduate School Trustsoft *
University of Oldenburg, Germany and
Chair for Software Design and Quality
University of Karlsruhe, Germany
{koziolek,sbecker,happe}@ipd.uka.de

**Abstract.** Performance predictions aim at increasing the quality of software architectures during design time. To enable such predictions, specifications of the performance properties of individual components within the architecture are required. However, the response times of a component might depend on its configuration in a specific setting and the data send to or retrieved from it. Many existing prediction approaches for component-based systems neglect these influences. This paper introduces extensions to a performance specification language for components, the Palladio Component Model, to model these influences. The model enables to predict response times of different architectural alternatives. A case study on a component-based architecture for a web portal validates the approach and shows that it is capable of supporting a design decision in this scenario.

## 1 Introduction

Performance problems in large distributed software systems, such as high response times and low throughput, often result from poor architectural decisions during early development stages [22, 16]. When they are discovered in a running system, they might require a costly redesign of the architecture and often cannot be fixed by simply revising small parts of the code. To avoid redesigns, software architectures should be analysed for their performance properties as early as possible.

As large software architectures are composed of (possibly third-party) components [23], architects need performance specifications of each individual component to conduct performance predictions for the planned architecture. However, specifying the performance of a software component is difficult, as component developers cannot make any assumptions on the context (i.e., underlying hardware, operating system, usage profile, performance of required services, etc.) of their components and thus need to specify the performance independently of the context. Many existing approaches for component-based performance prediction [2, 5, 11, 12, 8, 6, 25] neglect one or more

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

context dependencies in their component specifications. Especially the context-dependent configuration of components as well as the input and output parameters of component services as part of the usage profile are usually disregarded.

This paper extends our existing component specification language, the Palladio Component Model (PCM) [4], with constructs to model the performance influences by configuration parameters of a component as well as input and output parameters. Component developers may characterise a set of configuration parameters for their components (called component parameters), which can be adopted by software architects to a specific usage profile. For example, a performance-relevant component parameter might be the size of data the component uses for computation or a compression rate for a component compressing files.

We have implemented a transformation of an architectural model composed out of such component specifications into a formal analysis model, which can be used to predict response times for use cases of the architecture as probability distributions. Our approach aims at evaluating the architectures of large distributed systems. It does not try to make accurate predictions for real-time systems, but to guide design decisions for distributed systems at early development stages, where some performance-relevant details might still be unknown.

The contributions of this paper are i) extensions to an existing component performance specification language (PCM) for modelling component parameters as well as input and output parameters, ii) the implementation of a model transformation algorithm to automatically map component performance specifications to an analytical model, and iii) a proof-of-concept case study on a component-based architecture. We validate the applicability of our approach by comparing our predictions for different usage profiles of the same architecture with measurements from an implementation. Our results show that the approach is capable of validating a service level agreement in our scenario.

The paper is organised as follows: Section 2 surveys related work in the area of model-based performance prediction for software architectures. Section 3 introduces extensions to the Palladio Component Model, our specification language for component and architecture performance, and includes an example of the newly introduced concepts. Afterwards, Section 4 briefly describes the tool-supported transformation of the PCM into a stochastic process algebra, which is explained in more detail in Section 5. After listing assumptions and limitations of the approach, Section 6 contains a case study of performance predictions with our method for a larger component-based software architecture. Section 7 concludes the paper and points out future work.

## 2 Related Work

Model-based performance prediction methods for software architectures originate from the SPE (Software Performance Engineering) approach by Smith et al. [22]. They have been surveyed by Balsamo et al. [2] and, specifically for component-based systems, by Becker et al. [3].

Bondarev et al. [6] proposed a similar approach to ours, but aim at components in an embedded system environment. Dependencies between input parameters and resource usage of a component are modelled explicitly in this approach, but it is assumed that

input parameters can be characterised as constant values. Our approach allows stochastical characterisations of parameters, which is more accurate for the targeted domain of large distributed systems.

The CB-SPE approach [5] is based on component specifications, which are parameterisable for different resource environments (e.g., CPU time, bandwidth, memory buffer), but not for different usage profiles. It furthermore assumes that it is known which required services are invoked in a component upon calling a provided services. This dependency is modelled explicitly in our approach.

KLAPER [11] is an intermediate specification language for component performance and reliability aiming at reducing the effort for model transformations. It allows the specifications of input parameters, but does not contain component configuration parameters or output parameters.

Hamlet et al. [12] aim at components resembling mathematical functions. They divide the input space of a component into several subdomains. The execution time for each subdomain is provided by the component developer, and the software architect needs to execute the component for each subdomain to deduce which other components are called.

Liu et al. [15] presented an approach for design-level performance prediction for systems based on the EJB-platform. They build a quantitative performance model from an application-independent performance profile of the underlying component container, which has be derived by measurements. This approach does not consider different roles in the development process and does not aim at systems incorporating components developed by third parties.

## 3 Design Model

Our approach for performance prediction follows the established separation between a design-oriented and an analytical model [2]. Developers create the design model during early development stages of a software system with an UML-like modelling language assisted by tools. It is then transformed via tools into a restricted stochastic process algebra, which is solved mathematically to reveal bottlenecks or violated performance requirements in the architecture. This approach hides the complexity and analytical concepts of the underlying algebra from the developers, thereby possibly enabling even non-performance specialists to manage a performance prediction.

### 3.1 Palladio Component Model

The design model used in our approach is the Palladio Component Model (PCM) [4]. It is a meta-model specified as an instance of MOF (Meta-Object Facility) [18] (similar to the UML meta-model) and can been seen as a domain-specific modelling language for QoS (Quality-of-Service) predictions, which has been aligned to different developer roles in CBSE.

Using the PCM, *component developers* specify their components' performance properties with so-called service effect specifications (cf. Section 3.2). *Software architects* compose these component specifications from repositories to build an application spec-

ification. *Deployers* model the middleware and hardware resources of the targeted application environment and allocate component specifications to these resource models. Finally, *business domain experts*, who are familiar with expected user behaviour, model the usage profile of the application.

The PCM provides a modelling language for each of these roles, and each language only includes concepts familiar to this role. This approach enables the division of work targeted by CBSE, and reduces the model's complexity for each role. Once all models are specified, they can be combined and then be transformed into the analytical model (cf. Section 4).

In the following, the paper will focus on the component specification language of the PCM, which has been specified with the Eclipse Modeling Framework (EMF) in Ecore [7]. A description of the other specification languages (e.g., component assembly model, resource model) can be found in [20]. Software components are black box entities with contractually specified interfaces [23]. In the PCM, components are either composite components, which are assembled out of inner components, or basic components, which cannot be further decomposed. Interfaces are first-class entities in the PCM and can be associated to a component in a providing or requiring role. An interface consists of a number of service signatures, which contain a list of parameters, a return type, and a list of exceptions.

### 3.2 Service Effect Specification

To specify the performance properties of a component's service, our approach uses service effect specifications (SEFF) [4, 21], which describe how a provided service calls required services and how it uses system resources. Such a specification makes the dependencies between resource usage and input parameters explicit, because component developers cannot know in advance how the component will be used by third parties. The SEFF is a strong abstraction from the service's source code including only performance-relevant information and thus does not violate the black box principle.

The SEFF-metamodel in the PCM is shown in Fig. 1-3. A `BasicComponent` may contain a number of `ServiceEffectSpecifications` (Fig. 1), which each reference a signature of an associated provided interface. For performance predictions, our approach uses `ResourceDemandingSEFFs`, which contain a number of `Abstract-Actions`.

**Component Parameters** As components are often implemented with object-oriented techniques, they can have an internal state at runtime, which can be the result of former service calls, constructor calls, or other forms of configuring the component (e.g., via deployment descriptors). In this paper, the elements of this state are called *component parameters*, as they can be accessed from any service of the component and are not local to a specific service. Essentially, component parameters extend the input space of the component's services. If they influence the performance of a component significantly, a characterisation of their values should be included in the performance specification.

Following the suggestions in [12] component parameters are treated as additional input parameters of a service. To avoid a state-space explosion in our model, we consider these values as unchangeable during service execution, which is a simplification

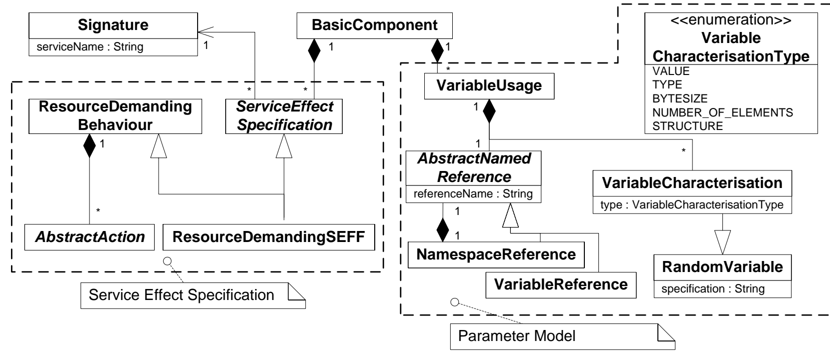



**Fig. 1.** Basic Component, Service Effect Specification, and Component Parameters

but nevertheless covers a number of practical situations and is often sufficient to analyse single use cases. Furthermore, we assume that each client of the component's services accesses the same values in a specific use case, i.e., there are no client-specific component parameters.

Therefore, a `BasicComponent` may contain a number of `VariableUsages` (Fig. 1) to model component parameters, which in turn contain a name for the variable (`AbstractNamedReference`) and a description of its actual values (`VariableCharacterisation`). Variables can be characterised for their value, type, or size in bytes in the PCM, as all these properties may influence performance. Additionally, the number of elements and structure of collections can be characterised (see [14]). Variable characterisations are constant or discrete `RandomVariables` enabling the use of probability distributions, which is useful for characterising larger user groups with different habits. A variable may have multiple characterisations of different types (e.g., VALUE and TYPE at the same time). Composite data structures can be characterised by using multiple `VariableUsages` with the same `NamespaceReference` and different inner `VariableReferences` (e.g., "customer.name" and "customer.cash"). Only parameters influencing performance properties need to be included into SEFFs, all other parameters can be abstracted.

As an example of a component parameter, a component compressing files could contain a parameter compressionRatio, whose value domain ('high', 'medium', 'low') is characterised with a probability mass function (PMF) assigning a probability to each of the values (e.g., 10% 'high', 20% 'medium', 70% 'low'). Depending on the configured compression ratio and assuming fixed-size data, the response time of calling the component's services would change, as a high ratio would result in slow calls, whereas a low ratio would lead to faster executions. The component developer can specify this parameter in the component description and possibly also provide a default value (e.g., 100% medium). Upon including the component in an architecture, the value can be changed by domain experts or software architects.

**Actions, Input- and Output-Parameters** Fig. 2 shows the different specialisations of `AbstractAction` from Fig. 1. Actions in SEFFs can either be `ExternalCallActions` referencing required services or `AbstractResourceDemandingAc-`

tions describing internal executions, which use the resources the component is deployed on. Actions are arranged in a chain, as each action references a predecessor and successor. The chain starts with a StartAction, then might contain internal executions, branches, loops, or forks (described later), and ends with a StopAction. AcquireAction and ReleaseAction allow the acquisition of passive resources, such as threads or semaphores.

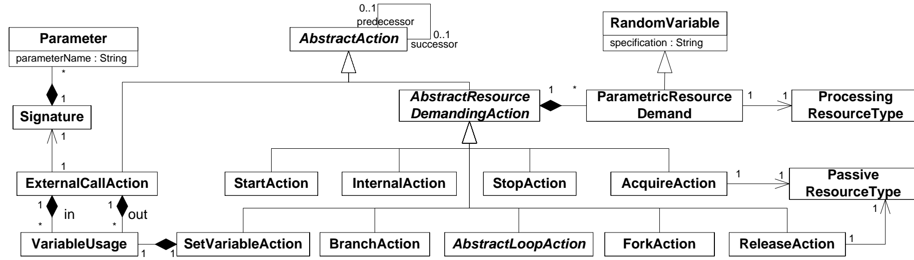



**Fig. 2.** Actions in Service Effect Specifications

ExternalCallActions contain variable usages to characterise the input parameters when calling a required service. Also, they allow to assign characterisations of output parameters from required service calls to local variables. Afterwards, the local variables can be referenced by following actions. The characterisations of input and output parameters are VariableCharacterisations, the same modelling entity as for component parameters. They might themselves depend on the input parameters of the specified service, for example if a parameter is processed internally and then passed to an external service (e.g., extCallInput.VALUE = SEFFInput.VALUE). To express changes on the characterisation made by the service, our framework for stochastical expressions allows arithmetric operations on the corresponding random variables such as additions, subtractions, multiplication etc. For example, a service may add some information to a file and then pass it to another service (e.g., extCallInput.-BYTESIZE = SEFFInput.BYTESIZE + 100, where the byte sizes are specified as random variables.).

We introduce a new action (SetVariableAction) to abstractly characterise the return values of a service in a SEFF. It references a VariableUsage, which in turn can include the name of a return value or of an output parameter. The characterisation of these VariableUsages might again use input parameters of the SEFF. If multiple SetVariableActions are specified in different branches of the SEFFs, the characterisation of the corresponding output parameter is weighted according to the (possibly nested) branching probabilities. Using a SetVariableAction within a loop results in a characterisation of the output parameter weighted with the probabilities for the number of loop iterations.

To consume the resources the service is deployed on, all actions other than ExternalCallActions can contain a ParametricResourceDemand. As component developers can and should not know the concrete resources the component will be deployed on by third parties, resource demands are specified for Processing-

ResourceTypes. These types can be for instance CPU, hard disk, network device, etc. for which component developers specify the demand in terms of CPU cycles needed, bytes read from hard disk, bytes sent over the network, etc. as random variables. Once deployers specify the processing rates of concrete resources of these types, timing values can be derived from the resource demands. Resource demands may depend on input parameters, and this dependency can be specified using the same stochastical expressions as decribed above.

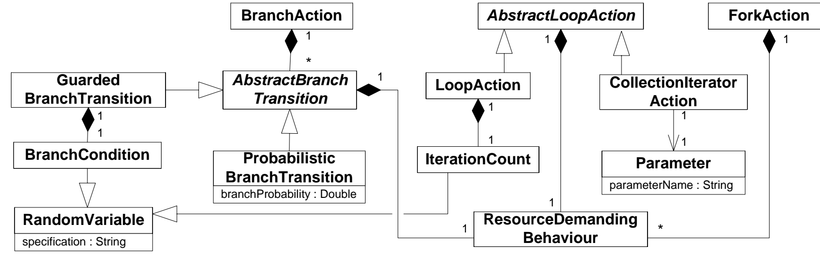



**Fig. 3.** Control Flow in Service Effect Specifications

**Control Flow** Fig. 3 illustrates the control flow operations in the SEFF-metamodel. The behaviour of a SEFF may include branches, loops, and forks. BranchActions split the control flow with an OR-semantic, only one of the following branch transitions is executed, while ForkActions split the control flow with an AND-semantic, i.e., each of its inner behaviours is executed concurrently.

Branch transitions can be guarded or probabilistic. In the former case, the component developer specifies a boolean random variable (BranchCondition) in relation to an input parameter of the SEFF. As our framework for stochastical expressions allows compare-operations such as equals, less, greater, etc. and also AND/OR combinations complex constraints on the input parameters for executing a branch can be expressed. The constraints shall always span the whole value domain of a parameter (e.g. $X < 10$ and $X >= 10$), and shall not intersect.

In the case of ProbabilisticBranchTransitions, the component developer specifies a probability for executing a specific branch. Though the behaviour of component services is usually not probabilistic, it might occur in complex services that a constraint on the input parameters for executing a branch cannot be specified easily. In this case, the component developer can specify just a probability.

AbstractLoopActions can either be LoopActions, which include an integer random variable for the number of loop iterations (IterationCount), or CollectionIteratorActions, which iterate over the elements of a collection provided as an input parameter to the service. In that case, the number of iterations is the same as the number of elements in the collection, which might have been specified as a probability distribution.

LoopActions assume the stochastical independence of parameters used in the loop body to reduce the necessary computations. Opposed to this, Collection-IteratorActions assume a stochastical dependency between the characterisations of the inner elements of the collections used in the loop body. If a characterisation of an

inner element of the collection is used a second time within the loop body, its characterisation is stochastical dependent to its first use. For example, an integer variable might adopt the values 1 and 2 with a certain probability. If the random variable was evaluated to 1 on the first use of the variable, it also has to evaluate to 1 on the second occurrence. Including this stochastical dependency increases the computational complexity, but leads to more accurate predictions.

### 3.3 Example

Fig. 4 shows an example instance of the SEFF including most of the concepts introduced above. The concrete syntax is an UML activity and stereotype actions and annotations with the classes from our metamodel. Component1 provides Interface1, which includes the signature of service1 with two input variables X,Y as well as the return type Collection. Furthermore, a component parameter Z is specified for the component, and its bytesize is characterised with a constant (2000).

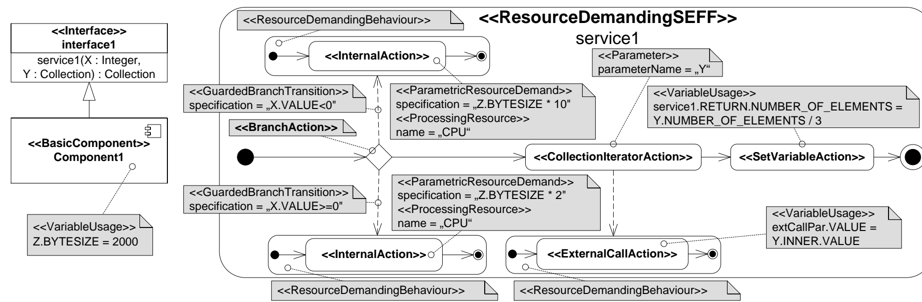



**Fig. 4.** Service Effect Specification

After the start action, the SEFF contains a branch, which includes two `GuardedBranchTransitions` defining a constraint on the value of input parameter X. Once the value distribution for X is specified by the domain expert, the probability for the guards to become true can be determined. Both branched `ResourceDemandingBehaviours` include an `InternalAction`, whose resource demand is specified in dependency to the bytesize of component parameter Z. After the branch, the service iterates over the elements of input collection Y with the `CollectionIteratorAction`. Within the `ResourceDemaningBehaviour` of this loop, an external service is called, and its input parameter extCallPar is characterised in dependency to the inner elements of input collection parameter Y. Finally, the SEFF characterises the number of elements of service1's return value in dependency of input collection Y.

Note, that the SEFF only contains performance-relevant information, such as resource demands, abstract control flow, and calls to external services. The actual code of the component can contain an additional amount of internal computations, which are not performance-relevant, and thus have been abstracted.

## 4 Transformation

In addition to the SEFFs by the component developers, the other three roles need to specify their model instances (i.e., usage model, resource environment model, assem-

bly model, allocation model, etc.) in order to create a full PCM instance. Once all models are specified and combined, they can be transformed into our analytical model. For the specification of the model instances, we have implemented graphical editors using Eclipse GEF/GMF [17]. The transformation itself as well as the evaluation of our analytical model have been implemented as Java applications. For the future, it is planned to use a standardised model-to-model transformation language such as QVT [19].

The transformation consists of two steps: First, the parametric dependencies in the SEFFs are resolved, then the PCM instance is converted into an instance of our stochastic process algebra. A brief description of both steps follows.

In the first transformation step, usage model and SEFFs invoked by it are traversed with a visitor (generated by EMF) and decorated with so-called usage contexts and actual allocation contexts at each invocation of a SEFF (Fig. 5). An usage contexts results from solving parametric dependencies on branch transitions, loop counts, and variable usages. For example, if a branch transition contains a guard $X < 10$, meaning that the branch would be executed if input variable $X$ was smaller than 10, and from the usage model it is derived that $X$ is always 5, then the corresponding `BranchProbability` in the usage context will hold a probability of 1.0 after this step.

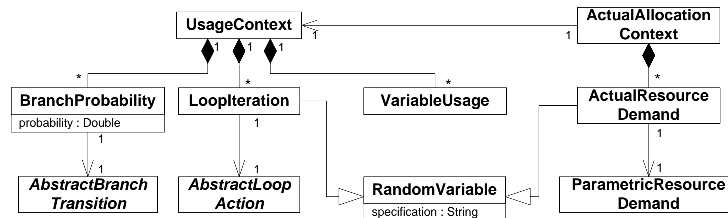



**Fig. 5.** Usage Context, Actual Allocation Context (Metamodel)

An actual allocation context contains timing values (`ActualResourceDemands`), which have been derived by solving the dependencies on parametric resource demands. Note that classes from the SEFF metamodel are only referenced *from* the context models and not in the other direction, so that a SEFF specification is independent from a concrete context. As an example, Fig. 6 shows a set of variable characterisations on the left side and the SEFF from the former section with solved parametric dependencies and decorated contexts on the right side.

In the second transformation step, the PCM instance (now including the derived context models) is again traversed with a visitor to create a simplified representation in our stochastic process algebra. An instance of the algebra is represented as a binary tree, whose inner nodes represent the operations sequence, alternative, loop, and parallel execution, and whose leafs contain timing values as probability density functions (PDF). The second transformation step merges a usage scenario with the participating SEFFs, and transforms the n-ary tree of actions in the PCM into the binary tree of the algebra. Otherwise the mapping from constructs in the PCM instance and the context model to constructs of the algebra is straightforward.

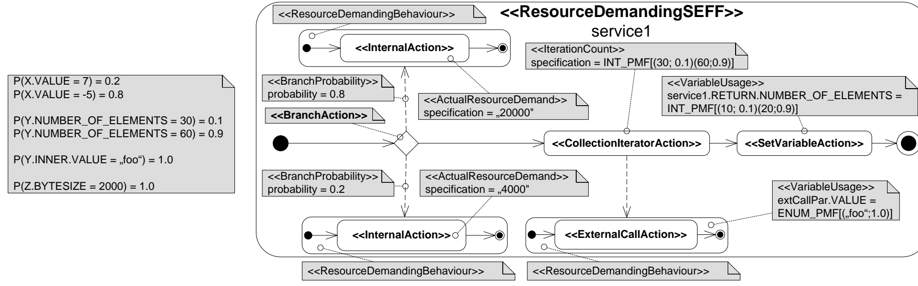



**Fig. 6.** Service Effect Specification with Usage Contexts, Actual Allocations Contexts

## 5 Analytical Model

The computation of a service's execution time uses a stochastic process algebra (SPA) based on regular expressions. Its grammar is defined by the following BNF:

$$P := a \ | \ P \cdot Q \ | \ P +_\pi Q \ | \ P^{*(l)}$$

For the scope of this paper, we use the common semantics of regular expressions enriching it with a semantic for timed behaviour. Opposed to process algebras in general, recursive behaviour is forbidden here, instead our SPA uses the less expressive construct of loops. This allows us to perform a relatively straightforward analysis of the described systems. The complete version of the process algebra also supports parallel processes and synchronisation (preliminary version can be found in [13]). In the following, the time semantics of the rules above will be explained. Symbols will be denoted by small letters $(a, b)$ and processes by large letters $(P, Q)$.

### 5.1 Computations

The execution time of a **symbol** $a$ is a random variable $X_a$ characterised by its PDF $f_a(t)$. In our model, arbitrary distribution functions are allowed, which are assumed to be independent and identically distributed (iid, see Section 5.2). $X_a$ specifies the time passed while processing $a$. If $X_t$ is the time when $a \cdot P$ starts, the processing time of $a$ is added to $X_t$ when it finishes, so $X_t' := X_t + X_a$, and $a \cdot P$ then behaves like process $P$ at time $X_t'$.

As for symbols, the execution time of process $P$ is denoted by an iid random variable $X_P$ characterised by PDF $f_P(t)$. The execution time of a **sequence** of two processes $P$ and $Q$, $P \cdot Q$ is the sum of their execution times $X_{P \cdot Q} = X_p + X_q$. Since $X_P$ and $X_Q$ are assumed to be iid, their characterising PDFs can be convoluted

$$f_{P \cdot Q}(t) = (f_p \circledast f_q)(t),$$

where $\circledast$ denotes the symbol for convolution.

The probabilistic choice or **alternative** of two processes $P$ and $Q$, $P +_\pi Q$, either has the execution time of process $P$ with probability $\pi$ or of process $Q$ with probability $(1 - \pi)$. To define this behaviour, the uniform distribution between zero and one $u(x)$

is used, where $u(\cdot)$ denotes drawing a sample from $u(x)$. Let $u = u(\cdot)$ be one sample of PDF $u(x)$, then

$$X_{P+_\pi Q} = \begin{cases} X_P, & \text{if } 0 \leqslant u < \pi \\ X_Q, & \text{if } \pi \leqslant u \leqslant 1 \end{cases} .$$

The PDF of the alternative is the weighted sum of the single PDFs

$$f_{P+_\pi Q}(t) = \pi f_P(t) + (1-\pi) f_Q(t)$$

If process $P$ is executed in a **loop**, $P^{*(l)}$, the number of loop iterations is specified by a probability mass function (PMF) $p_l(i) = P_l(X = i)$ denoting the probability that process $P$ is executed $i$ times in a row. Then, if $u = u(\cdot)$ is a sample of the uniform distribution $u(x)$ and $F_l(x)$ is the cumulative distribution function of $p_l(i)$, the execution time of a loop is

$$X_{P*^{(l)}} = \begin{cases} 0 & 0 \leqslant u < F_l(0) \\ X_{P,1} & F_l(0) \leqslant u < F_l(1) \\ X_{P,1} + X_{P,2} & F_l(1) \leqslant u < F_l(2) \\ & \vdots \\ X_{P,1} + X_{P,2} + \ldots + X_{P,N} & F_l(N-1) \leqslant u < F_l(N) \end{cases}$$

where $N \in \mathbb{N}_+$ is last value with $p_l(N) > 0$ and $p_l(i) = 0 \quad \forall i \in \mathbb{N}_+, i > N$. $X_{P,i}$ is the $i$th instance of random variable $X_P$. The PDF of a loop can be computed by

$$f_{P*^{(l)}}(t) = \sum_{i=0}^{N} p_l(i) \left( \circledast_{j=1}^{i} f_P \right)(t)$$

Tools and techniques from signal processing efficiently compute the execution time of a process. We approximate continuous PDFs with discrete PMFs using a predefined sampling rate as described in [9].

### 5.2 Assumptions and Limitations

The following section briefly describes assumptions and limitations underlying our approach. More detail can be found in [14, 4].

**Independent and identically distributed random variables:** Random variables characterising resource demand are assumed to be stochastically independent. This might not hold in some realistic cases, for example, if a resource is overloaded and all resource demands for this resource will lead to slow execution times. Furthermore, PDFs for resource demands are assumed to not change over time (e.g., to model a warm-up phase and a normal operation phase of the system).

**Markov property on branches:** As our SPA is based on Markov chains, our approach inherits their usual assumptions. For loops, the Markov property (the probability of going from one state to another is independent of the former execution path) has been weakened, as our SPA allows the specification of PMFs for the number of loop iterations and is not bound to a geometrical distribution. For branches, the Markov property is still present, but the SEFF-model contains branch conditions in dependency to input parameters.

**Single-user analysis:** As a first step, the analysis model assumes that only one user executes a scenario at a time. However, the results from our approach can be fed into existing tools for queueing networks to obtain predictions in a multi-user setting. Additionally, in [4] we have introduced a simulation tool for the PCM capable of simulating multiple concurrent users.

**Availability of models:** SEFFs for each component in the architecture must exist to apply our method. If new components are designed, it is possible to derive the SEFF specifications from the design documents and publish them in a component repository. For already existing components, we are working on code analysis tools for component developers, to derive SEFFs semi-automatically out of source code, so that legacy components can be integrated into our approach.

**Static architecture:** The PCM does not support dynamic component architectures, where new components can be created or the links between components can change at runtime. It is assumed that the set of components and their connection in the architecture are fixed for our performance prediction.

## 6 Case Study

We have conducted a proof-of-concept case study on a component-based architecture comparing response time predictions based on our models with response time measurements made with an implementation of the architecture. As the newly introduced component specifications allow evaluating an architecture under different usage profiles, we examined the architecture with two different usage profiles.

Specifically, the questions before the case study were: Can our method predict correctly, whether this component-based architecture can meet a service level agreement (SLA) under two different usage profiles? How large is the prediction error of our method in the measured usage scenarios?

### 6.1 Architecture, Scenario & Usage Profiles

The case study analyses the `MediaStore` architecture, a web shop for music and video files modelled according to the functionality of the iTunes Music Store. It is a three-tier architecture (Fig. 7) with a client tier, an application server hosting components implementing the business logic, and a database tier with two MySQL databases connected to the application server via Gigabit Ethernet. For the application server tier, we chose the open-source variant of Sun's Glassfish application server [10] which is fully EJB3 compliant.

In the usage scenario of our case study, a user downloads a set of files (e.g., a music album) from the store. Therefore, the user provides the `WebGUI`-component with a query resulting in a set of music titles, which is forwarded to the `MediaStore`-component. The database `AudioDB` is searched for the files, which afterwards get transferred from the database server via the network connection to the application server. Finally, the client can download the files. The `UserDB` database is not used in this scenario.

The filling degree of the `AudioDB` is a performance-influencing factor (e.g., when executing queries over the contained tables) and has been exposed with a collection

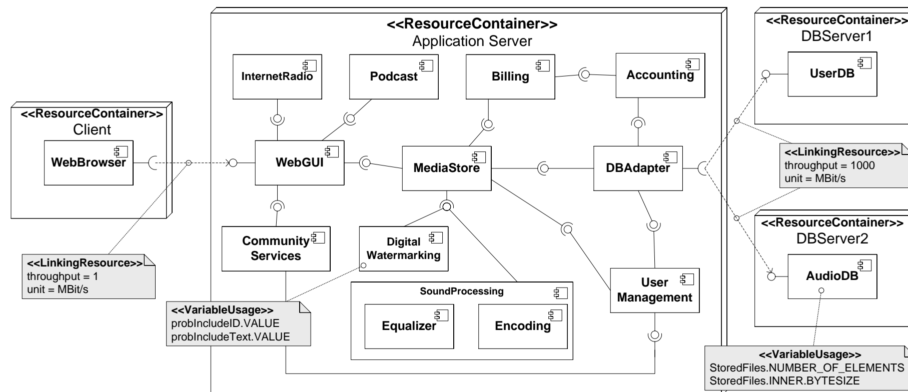



**Fig. 7.** Media Store: Static/Allocation View

component parameter (`StoredFiles.NUMBER_OF_ELEMENTS`). The actual value of this parameter depends on the context the component is used in. Furthermore, the size of the files stored on the database server influences the transmission delay between database server and application server, and is also modelled as a component parameter (`StoredFiles.INNER.BYTESIZE`).

As a measure for copy protection, we incorporate a component `DigitalWatermarking` into the architecture. It is able to unrecognisably watermark individual media files with additional information. This component can be configured to watermark media files with the current user ID, so that the user could be tracked down, should the file appear somewhere on the Internet. This configuration option has been modelled as a component parameter `probIncludeID`, which specifies the probability of including the ID. Additionally, the component can be configured to include additional texts like lyrics or subtitles into the files, which has been modelled with the component parameter `probIncludeText`.

Our performance prediction method shall check, whether the time between issuing the download request and starting the download is sufficiently short. From the requirements, there is a service level agreement (SLA) of at least 90% of calls returning in under 8 seconds, which has to be met even after watermarking is introduced.

The performance influencing information of the component services involved in the case study has been modelled using SEFFs illustrated in Fig. 8. The parameteric dependencies within the SEFFs have been derived by monitoring the components and analysing the results using statistical regression techniques. This step has to be performed once for each component by its developer. Using such annotated SEFFs, different software architects can assemble the components to individual architectures and predict the performance under their individual usage profiles.

As an example, Fig. 9 illustrates the series of measurements and the linear regression for searching the database, if the number of files stored within changes. We used the derived values to specify the parametric dependency for the `InternalAction` ”search” of the service AudioDB.getfiles in Fig. 8.

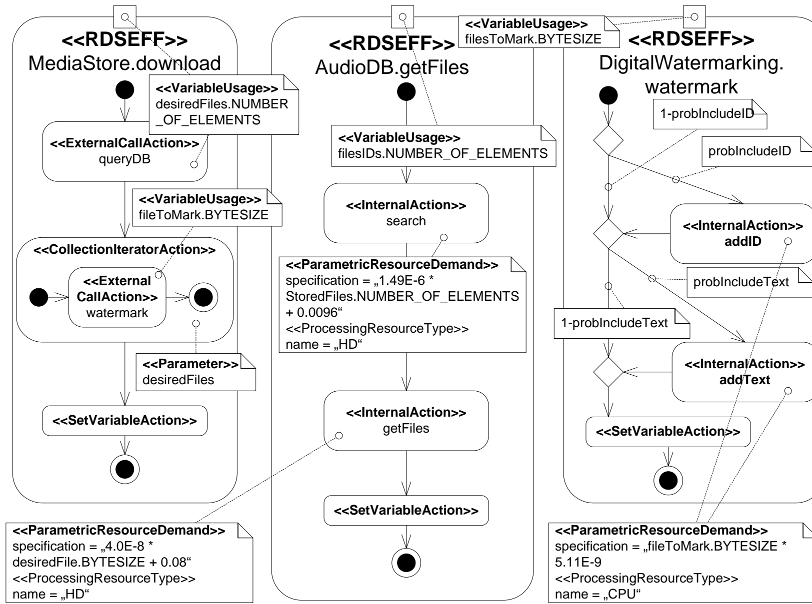



**Fig. 8.** MediaStore: Service Effect Specifications

Note, that this step of deriving parametric dependencies also has to be done parameterised by the hardware and middleware layer, on which the components are deployed on. However, this is out of the scope of this paper.

The case study considers two different settings. In `Setting1`, the components and the `MediaStore` architecture are used to build a music store, in which users usually request 10-14 files (a music album, number of files uniformly distributed) with their filesizes distributed as given in Fig 10. The database is filled with 250.000 entries of different music titles (i.e., `AudioDB.StoredFiles.NUMBER_OF_ELEMENTS` = 250000). Only the user ID branding from the watermarking component is used in this setting (i.e., `probIncludeID`=1.0, `probIncludeText`=0.0).

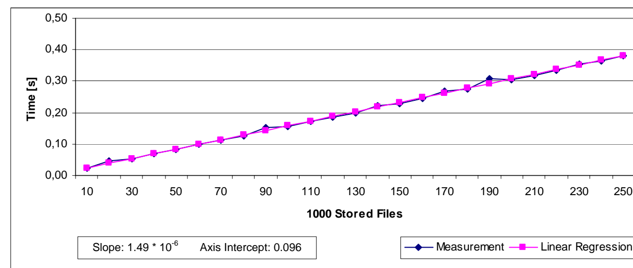



**Fig. 9.** Time for Searching the Database

| 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 |

**Fig. 10.** File size distribution

In `Setting2`, the components and the architecture are used to build a video store, where users usually only request a single file (a movie), whose size is uniformly distributed between 95 and 105 MB. The database is filled with 10.000 entries of different movies (`AudioDB.StoredFiles.NUMBER_OF_ELEMENTS` = 10000). The software architect has configured the watermarking component to include the user ID and also subtitles for the movies (i.e., `probIncludeID`=1.0, `probIncludeText`=1.0).

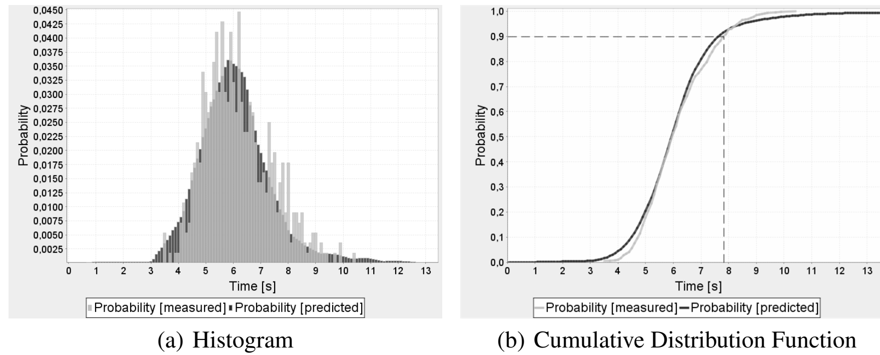
For the implementation of the `MediaStore`, we exploit the architectural model we have used for the predictions. We have implemented a Model-2-Text transformation on the PCM model instance, which generates code skeletons implementing the components as EJB3. The code skeletons already contain the control flow parts of the SEFFs (i.e., loops, branches, etc.), only the logic of internal actions is missing and has been implemented manually. Additionally, build scripts, deployment descriptors, configuration files, and a test client for performing the response time measurements are generated. We introduce measurement probes into the implementation using aspects, which we weave into the code using AspectJ [1]. Finally, we have ensured in a pre-test run that the measurement probes do not distort the measured response times significantly. We have measured the response time in both settings approx. 500 times to get distribution functions.

### 6.2 Results

The predictions and measurements for the two settings of our case study can be found in Fig. 11- 12. The figures illustrate both predictions (dark grey) and measurements (light grey) as histograms and cumulative distribution functions (CDF), which are placed on top of each other to enable comparing them visually. Matching areas of both functions are shown in medium grey.

For `Setting1`, the probability functions widely overlap (Fig. 11(a)). The most probable predictions for the response time is at around 6 seconds, which matches the measurements quite accurately. To test for goodness of fit, we used a Kolmogorov-Smirnov-Test [**?**] (KS-test), which was not able to reject our null hypothesis of prediction and measurement having the same underlying probability distribution at a significance level of $\alpha = 0.01$. From the CDF it can be derived that it was predicted that the SLA of 90% of the calls returning in less than 8 seconds could be met (Fig. 11(b)). This prediction also matched the measured values, where actually 92% of the calls returned in less than 8 seconds. The deviation between prediction and measurement at the 90% mark (7.7 seconds vs. 7.8 seconds) is approx. 0.1 seconds, i.e., a difference of 1.3 percent.

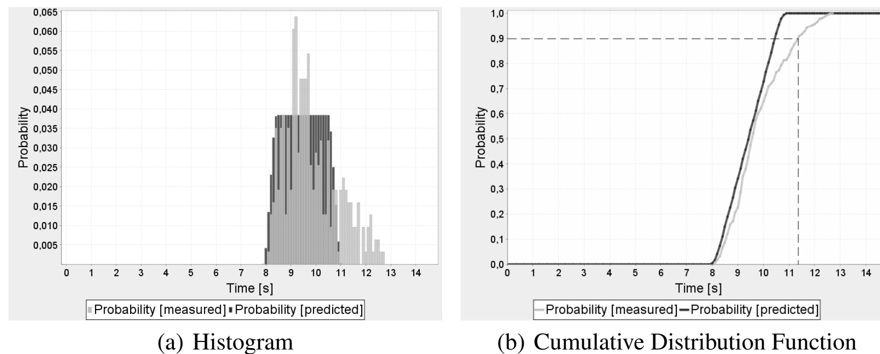
In `Setting2`, the probability functions overlap to a large extent, but the measured response times spread further than the predicted ones (Fig. 12(a)). This is due to the higher network load because of the larger file sizes, which leads to a higher distortion of the measurements [24]. However, the most probable predicted and measured values are both around 9.5 seconds. As in setting1, the KS-test was not able to reject our null




(a) Histogram (b) Cumulative Distribution Function

**Fig. 11.** Response Times `Setting1`

hypothesis. In this case, it was predicted that 90% of the calls returned in 10.4 seconds, which would violate the required SLA of 8 seconds (Fig. 12(b)). The measurements confirmed that under `Setting2` the SLA would indeed be violated, as the 90% of the calls returned in only 11.3 seconds. The error between prediction and measurement was 0.9 seconds or 8.0 percent.




(a) Histogram (b) Cumulative Distribution Function

**Fig. 12.** Response Times `Setting2`

Concludingly, our approach correctly predicted the compliance or violation of the required SLA in both studied cases. In these cases the pointwise error between prediction and measurement was below 10 percent.

## 7 Conclusions and Future Work

This paper has introduced extensions to an existing component specifications language to model characterisations of component parameters as well as service input and output parameters. Such parameters can influence the resource usage or control flow of a component service significantly and should thus be included into QoS-predictions. We have described the fully automated model transformation, which maps an architectural model of our component specifications to a stochastic process algebra. Solving the algebra analytically has yielded response times for use cases of the system as distribution functions.

Component developers as well as software architects may benefit from this approach. Component developers can specify the performance properties of their components without exposing intellectual properties, and thereby increase their re-usability as users can quickly check the actual performance of the component in their environment. With the performance specifications, software architects can quickly evaluate different design decisions regarding the components used in their systems. By adjusting the parameters of the introduced models, they can check if performance requirements can be met under different system configurations.

Future work is directed at lowering the limiting assumptions of our approach. We are currently implementing code-analyses tools to semi-automatically derive SEFFs from existing legacy software components. This might reduce the effort for manually creating the models needed for the performance prediction. The concurrency modelling of our process algebra is still limited and needs to be extended and evaluated on multiprocessor systems. We plan to better include characteristics and configuration settings of the middleware into our approach. In the long term, we will extend our approach for dynamic component architectures, where the bindings between components can change at runtime.

## References


1. The AspectJ Homepage. http://www.eclipse.org/aspectj/.
2. Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-Based Performance Prediction in Software Development: A Survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, May 2004.
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 *Proceedings of the 6th International Workshop on Software and Performance (WOSP2007)*. ACM Sigsoft, February5–8 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 2004), Edinburgh, UK*, volume 3054 of *Lecture Notes in Computer Science*, 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 *Proceedings of the 31th EUROMICRO Conference (EUROMICRO'05)*, 2005.
7. Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework*. Eclipse Series. Prentice Hall, August 2003.
8. Evgeni Eskenazi, Alexandre Fioukov, and Dieter Hammer. Performance prediction for component compositions. In *Proceedings of the 7th International Symposium on Component-based Software Engineering (CBSE7)*, 2004.
9. Viktoria Firus, Steffen Becker, and Jens Happe. Parametric Performance Contracts for QML-specified Software Components. In *Formal Foundations of Embedded Software and Component-based Software Architectures (FESCA)*, volume 141 of *Electronic Notes in Theoretical Computer Science*, pages 73–90. ETAPS 2005, 2005.
10. GlassFish Open Source Java EE 5 Application Server. https://glassfish.dev.java.net/.

11. 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 *WOSP '05: Proceedings of the 5th international workshop on Software and performance*, pages 25–36, New York, NY, USA, 2005. ACM Press.
12. Dick Hamlet, Dave Mason, and Denise Woit. *Component-Based Software Development: Case Studies*, volume 1 of *Series on Component-Based Software Development*, chapter Properties of Software Systems Synthesized from Components, pages 129–159. World Scientific Publishing Company, March 2004.
13. 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, *Proceedings of the 3rd International Workshop on Formal Aspects of Component Software (FACS06), Prague, Czech Republic*, Electronical Notes in Computer Science, September 2006.
14. Heiko Koziolek, Jens Happe, and Steffen Becker. Parameter dependent performance specification of software components. In *Proceedings of the Second International Conference on Quality of Software Architectures (QoSA2006)*, volume 4214 of *Lecture Notes in Computer Science*, pages 163–179. Springer-Verlag, Berlin, Germany, July 2006.
15. Yan Liu, Alan Fekete, and Ian Gorton. Design-Level Performance Prediction of Component-Based Applications. *IEEE Transactions on Software Engineering*, 31(11):928–941, 2005.
16. D. A. Menasce, V. A. F. Almeida, and L. W. Dowdy. *Performance by Design*. Prentice Hall, 2004.
17. B. Moore, D. Dean, A. Gerber, G. Wagenknecht, and P. Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks. IBM, January 2004.
18. Object Management Group (OMG). Mof 2.0 core specification (formal/2006-01-01), 2006.
19. Object Management Group (OMG). Mof qvt final adopted specification (ptc/05-11-01), 2006.
20. Ralf H. Reussner, Steffen Becker, Jens Happe, Heiko Koziolek, Klaus Krogmann, and Michael Kuperberg. The Palladio Component Model. Technical report, Universitï¿½ Karlsruhe (TH), 2006.
21. Ralf H. Reussner, Heinz W. Schmidt, and Iman Poernomo. Reliability prediction for component-based software architectures. *Journal of Systems and Software – Special Issue of Software Architecture – Engineering Quality Attributes*, 66(3):241–252, 2003.
22. C. U. Smith and L. G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, 2002.
23. Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 2 edition, 2002.
24. T. Verdickt, B. Dhoedt, F. De Turck, and P. Demeester. Hybrid Performance Modeling Approach for Network Intensive Distributed Software. In *Proceedings of the 6th International Workshop on Software and Performance (WOSP2007)*, ACM Sigsoft Notes, pages 189–200, February 2007.
25. Xiuping Wu and Murray Woodside. Performance modeling from software components. *SIGSOFT Softw. Eng. Notes*, 29(1):290–301, 2004.