Parameter Dependent Performance Specifications of Software Components

Heiko Koziolek, Jens Happe, Steffen Becker

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

Abstract. Performance predictions based on design documents aim at improving the quality of software architectures. In component-based architectures, it is difficult to specify the performance of individual components, because it depends on the deployment context of a component, which may be unknown to its developers. The way components are used influences the perceived performance, but most performance prediction approaches neglect this influence. In this paper, we present a specification notation based on annotated UML diagrams to explicitly model the influence of parameters on the performance of a software component. The UML specifications are transformed into a stochastical model that allows the prediction of response times as distribution functions. Furthermore, we report on a case study performed on an online store. The results indicate that more accurate predictions could be obtained with the newly introduced specification and that the method was able to support a design decision on the architectural level in our scenario.

1 Introduction

Performance is an important quality attribute of a software architecture. It can by characterised by metrics such as response time, throughput, and resource utilisation. In many existing systems, the reason for bad performance is a poorly designed software architecture [15]. Performance predictions based on architectural descriptions of a software system can be performed before the implementation starts, thereby possibly reducing costs for subsequent refactorings to fix performance problems. It is the hope that such early analyses support the decision for design alternatives and reduce the risk of having to redesign the architecture after performance problems have been diagnosed in the implementation.

Component-based software architectures are well-suited for early performance predictions, if information needed for performance evaluation has been specified for each component by its developers. As component developers cannot know in which context their components will be deployed [8], these performance specifications should be parameterisable for different hardware resources, required services, and usage contexts to allow accurate predictions [3].

* This work is supported by the German Research Foundation (DFG), grant GRK 1076/1

In many performance prediction approaches, the component specifications are parameterisable for different usage contexts only by allowing to specify probabilities for the possible requests to the component's provided services (e.g., [4, 17]). It is often neglected that component services can be called with different parameters, and that these parameter can have a significant influence on the performance of the architecture.

The dependencies between parameters of a component service and its performance have to be made explicit by *component developers* in the specifications to allow accurate performance predictions. *System architects* can then adjust performance predictions to the expected usage profile. In some cases, the dependencies between parameters and performance might be intricate and hard to specify, for example if a service first performs complex computations on a parameter value an then changes its performance depending on the results. Furthermore, parameters might be complex objects or even other components, for which a reasonable specification is difficult. However, in this paper a first step to integrating parameters into performance specifications of software components shall be taken. Parameters considered here can be of a primitive or composite data type.

A notation based on extensions to the UML SPT profile [12] is provided to specify the dependencies between parameters and performance. This profile allows annotating UML diagrams with performance-related information. As many existing performance approaches already use this profile (e.g., [4, 11]), they could also benefit from the extensions presented in this paper. Tools evaluating annotated UML diagrams could be changed with low effort to incorporate the extensions. Another advantage of the UML-based notation is the familiarity of the developers, who often already know the UML language. However, the concepts underlying the approach presented here are not bound to the SPT profile and might be carried over to other notations (e.g., future performance related profiles).

The contribution of this paper is a modelling notation for parameter dependent performance specifications for software components and an according analytical performance prediction model. Unlike most performance prediction models, we explicitly incorporate the influence of parameters on resource demand as well as on the usage of external services in our predictions. A case study, in which response times for a design alternative of a component-based software architecture are predicted, is provided to illustrate the benefits of this approach.

The paper is organised as follows: Section 2 describes the modelling in our performance prediction approach and focuses on parameter dependencies. Section 3 shows the necessary computations, and Section 4 explicitly lists the assumptions underlying the approach. The case study of a performance prediction for an component-based online store is provided in Section 5. Section 6 points out related work, while Section 7 draws conclusions and sketches future work.

2 Modelling Component Performance

Several models from different developer roles are used for the prediction of the performance in a component-based architecture in our approach. The architecture itself is

modelled with a UML component diagram by the system architect. For each component, the component developers provide a specification, which will be described with more detail in the next paragraphs. Components are associated with system resources by system deployers using UML deployment diagrams. The user behaviour is modelled with activity diagrams, in which each action represents the invocation of an entry-level component service by a user.

All models are combined and completed by the system architect, who performs the predictions supported by tools. For this prediction approach, it is intended to use a development process model outlined in [9]. In the following paragraphs, component specifications, needed annotations, parameter characterisations, and modelling of resource demand are elaborated on.

2.1 Components and Service Effect Specifications

Software components are black-box entities with contractually specified interfaces. *Provided* interfaces publish services of the component. *Required* interfaces specify which external services are needed by the component. Moreover, in this approach a component specification contains a so-called *Service Effect Specification* (SEFF) for each provided service [13]. It describes how the provided service calls the services specified in the required interfaces. Here, a service effect specification is modelled with UML activity diagrams, where each action represents a call to a required service of the component. Activity diagrams are better suited for composition than sequence diagrams, as each external service call could be modelled with an additional activity diagram that is the SEFF of another component. In the example in Fig. 1, component C provides the service a and requires the services x, y, and z. The service effect specification for service a on the right hand side shows that, upon invocation, service a first calls service x, and then either service y or z before it finishes execution.

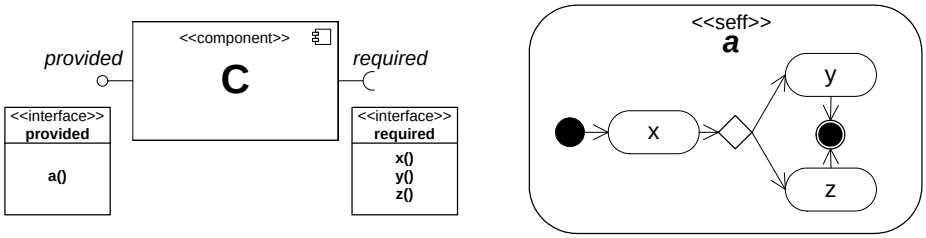


Fig. 1. Component C, Service Effect Specification for Service a

2.2 Annotations

To predict the performance of components, additional information is required in the specification, such as transition probabilities on branches, the number of iterations for

each loop, arrival rates of requests, and resource demands (i.e., the time an actions is expected to execute on a resource). This additional information is included into the SEFFs by annotations according to the UML SPT profile [12]. Each action and transition is annotated with the stereotype `<<PAstep>>`. The tagged values of the profile (e.g., to specify resource demands, transition probabilities, repetitions of steps etc.) can be used as described in the profile specification. Two extensions to the SPT profile notation are defined in the following (Tab. 1), to better reflect the influences of different usages of the architecture in the model.

Tag	Type	Multiplicity	Domain Attribute Name
PArep	PAloopValue	[0..*]	Step::repetition
PAparam	PApar	[0..*]	Step::parameter

Table 1. Redefined Tags

First, we redefine the tag `PArep` of the UML SPT profile allowing to either specify a mean value or to associate percentiles to the number of loop iterations. To ease the later analysis, loops are always modelled with the tag `PArep` here. Following the approach in [10], cycles within the activity diagrams indicated by backward transitions are not allowed. Loops have to be made explicit whenever they are used. This can be done in three ways. First, by annotating an arbitrary behaviour call node with a `PArep` tag and taking the called activity as loop body. Second, by using loop nodes provided in UML 2.0 as basic element. The loop node allows explicit modelling of the loop initialization, the repetition test and the loop body. Third, by using expansion areas in UML 2.0, which specify a set of actions that are executed iteratively on a collection of input objects. We use loop nodes and expansion areas in our examples later on.

```
<loopValue>          ::= ( <type-modifier>, <integer> )
<type-modifier>      ::= 'mean' | 'percentile', <real>
```

Tagged Value Type Definition 1: PAloopValue

Second, we define a new tag `PAparam` to characterise parameters of component services. The signature of a component's service specifies formal parameters. However, for QoS analyses, we need probabilistic characterisations of the actual parameters that the formal parameters can be bound on during runtime by the users.

2.3 Parameter Characterisation

Three forms of parameters can be distinguished. *Input* parameters are arguments passed to a provided service of a component. *Output* parameters are the return values of these services. *Internal* parameters can be global variables or configuration attributes of a

component. All these forms of parameters may have different influences on the performance of a component:

- **Resource Usage:** Parameters can influence the usage of the resources present in the system executing the component. For example, the time for the execution of a service that allows uploading files to a server depends on the size of the files that are passed as input parameters.
- **Control Flow between Components:** Service effect specifications describe how requests to provided services are propagated to other components. The transition probabilities or number of loop iterations in service effect specifications can depend on the parameters passed to a service.
- **Internal State:** Input parameters can be stored as global variables within a component, thus becoming internal parameters and altering the internal state of the component. Later, they can be used by computations of other provided services of the components and then influence resource usage or control-flow between components.

In the following, we consider primitive types (e.g., boolean, int, short, char) and composite types (e.g., String, List, Tree, Hash, Object). Other forms of parameters like streams or pointers are excluded here. It is useful to characterise parameters not only with constant values but with probability distributions. In the following specification of the tag `PAparam`, we allow modelling probabilities distributions over the value, the subtypes, the number of elements, the byte-size, and the structure of a parameters.

- **Value:** By providing a probability distribution for the value of a parameter, its input domain is partitioned into multiple subdomains. For example, for an integer-parameter x the domain can be partitioned into two subdomains with $x \leqslant 0$ and $x > 0$ depending on its influence on the performance of the component. The system architect can then specify a probability for each subdomain.
- **Subtypes:** Different subtypes can be passed to a service that has specified some supertype in the signature of a provided service. For example, a generic service drawing graphical objects might have a different response time depending on the type of objects passed to it (e.g., simple circle vs. complex polygon). In this case, it is useful to specify a probability distribution over the subtypes and to neglect the value of the parameters.
- **Elements:** For composite data types, it is more difficult to find subdomains over the value domain. The performance-influence of collections like array, tree, or hash can sometimes be characterised simply by the number of elements. Thus, it may be appropriate for such parameters to specify probability distributions over the number of elements.
- **Size:** Parameter values might also be passed between different servers thus creating a communication delay. The delay can best be analysed if the *byte-size* of the parameter is specified. To refine the specification a probability distribution for the size could be specified. Note that modelling the overall size of a parameter is appropriate if the inner structure of the parameter is unknown and the number of elements cannot be determined.

– **Structure:** Additionally, the *structure* of collections (sorted, well-formed, balanced, etc.) can have an influence on performance (e.g., presorted arrays are usually sorted quicker than unsorted arrays). Thus, a component developer could specify the behaviour of a component's service depending on the structure of a parameter passed to it.

For a single parameter, several of these characterisations could be specified. For more complex parameters like objects it might be necessary to first decompose them into more primitive types and then characterise these types. Besides specifying probability distributions, it may be convenient to specify mean values, constants, minimum or maximum values for parameters.

To model parameters in SEFFs, object nodes (represented as pins on actions) from UML 2.0 activity diagrams can be annotated with the newly defined tag `PAparam` (see tag definition 2). The string-value for `<paramValue>` is a representative characterisation of the subdomain of the value (e.g. "$0 <= value <= 10$") or can be used to specify a constant value (e.g. "foo"). The integer-value for `<paramSize>` specifies the number of bytes of the parameter, while the integer-value for `<paramElements>` models the number of elements in a collection. Finally, the string-value of `<paramStructure>` can be used to make any statement about the structure of the values in collection (e.g. "presorted" or "unsorted"), if it has an impact on performance (e.g. in sort-operations).

```
<paramStr>          ::= ( <property-modifier> )

<property-modifier> ::= <paramValue> | <paramType> | <paramSize> |
                        <paramElements> | <paramStructure>

<paramValue>        ::= 'value', <valueModifier>, <string>
<paramType>         ::= 'type', <valueModifier>, <string>
<paramSize>         ::= 'size', <probModifier>, <integer>
<paramElements>     ::= 'elements', <probModifier>, <integer>
<paramStructure>    ::= 'structure', <string>

<valueModifier>     ::= 'const' | 'min' | 'max' | 'percentile', <real>
<probModifier>      ::= 'mean' | 'sigma' | 'kth-mom', <integer> | 'max' |
                        'min' | 'percentile', <real>
```

Tagged Value Type Definition 2: PAparam

The dependency between a parameter and resource usage or control flow between components can be specified by using scalar variables of the tagged value language (TVL) of the UML SPT profile. The same variable can be used in a `PAparam` tag and another tag (e.g., `PAprob`, `PArep`, `PAdemand`, etc.) to model a dependency.

For example, in Fig. 2, a collection is passed to service `a` as parameter `P1`. The component developer has specified that the number of loop iterations of the required service `y` is three times the number of elements ("3*$P1elements", where $P1elements is a scalar variable of the TVL) of the collection passed to service `a` in parameter `P1`. Once the system architect specifies the number of elements in the collections the ex-

pected users will pass to the service, the number of external calls is also specified via the dependency.

In the same example, the type of parameter P2 is integer. The component developer has specified that, depending on whether values of P2 are smaller or larger than 100, either the required service y or z are called, because the transition probabilities of to these service depend on the variable specified for the parameter. The transition probability from x to z is specified as a difference to turn the cumulative percentiles into a probability.

Furthermore, parameter P3 is a binary large object. The component developer has specified that service a takes this parameter and returns parameter P4, which is 100 Bytes larger than parameter P4.

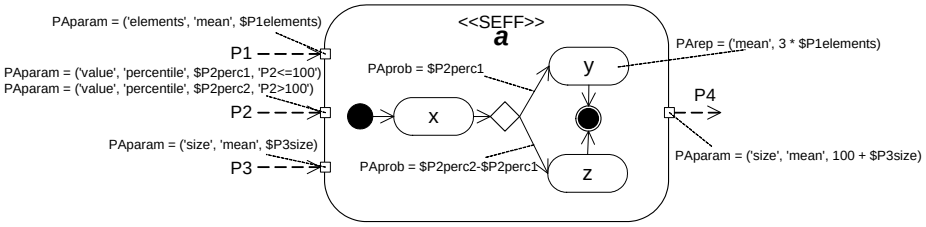


Fig. 2. Annotated Service Effect Specification for Service a including parameters

Note, that it is only necessary to characterise parameters if they indeed influence the performance. Most parameters do *not* change resource usage or alter the control flow between components, and their characterisation can be omitted. Characterising every parameter of the services in a complex component-based architecture would require too much effort and not support performance analysis. What parameters have to be characterised because of their influence on performance has to be defined by the component developer. So far, this task has to be done manually. However, it is conceivable to develop tools for the reverse-engineering of existing components to help component developers in obtaining the necessary specifications in a semi-automatic way. This is part of our future work.

3 Computing Component Performance

To calculate the response time for a service invocation, the resource demand of the service itself and the resource demands of required services have to be added. Resource demands are specified as probability mass functions (PMF) in our approach for a more refined modelling, and the necessary computations for combining these functions are described in the following. More detailed description of the computations can be found in [7, 10].

To conduct the computations, first the annotated activity diagrams are transformed into stochastic regular expressions, which are described in [10]. The mapping is straightforward: sequential executions are mapped to sequential expressions, control flow branches are mapped to alternative expression, and loops are mapped to single expressions with a distributions function for the number of iterations. The PMFs modelling the resource demand are annotated to each expression. So far, forks and join in activity diagrams are not supported by this approach. The abstract syntax tree of the resulting stochastic regular expressions is then traversed and the following computations are performed for each control flow primitive.

Sequence: The PMF for successive service invocations can be computed as the convolution of the single PMFs:

$$x_{R_1 \cdot R_2}(n) = x_{R_1} \circledast x_{R_2}[n]$$

Alternative: The PMF for a branch in the control flow can be computed as the sum of the PMF weighted by the transition probabilities:

$$x_{R_1 + R_2}(n) = p_1 x_{R_1}[n] + p_2 x_{R_2}[n] \text{ (average case)}$$

Loop: As we have also specified a PMF for the number of loop iterations with the `PArep` tag, the random variable for the resource demand of a loop is ($l(n)$ is the PMF for the number of loop iterations):

$$X_{R^l} = \begin{cases} X_R & \text{with probability} \quad l(1) \\ X_R + X_R & \text{with probability} \quad l(2) \\ \vdots & \\ \sum_{i=1}^{N} X_R & \text{with probability} \quad l(N) \end{cases}$$

with $N \in \mathbb{N}_0$ and $\forall i \geqslant N : l(i) = 0$. The corresponding PMF for the loop has the form:

$$x_{R^l}(n) = \sum_{i=1}^{N} l(i) \overset{i}{\underset{j=1}{\circledast}} x_R[n]$$

For the computation of the convolutions of the PMF, we use discrete Fourier transformations (also see [7]). The total response time obtained by analysing a stochastic regular expression can be fed into performance models like queueing networks to calculate the response time of a service in presence of multiple users in the system. These models also include contention delays of the requests into the response time.

4 Underlying Assumptions

The limiting assumptions of the prediction model concern the availabilty of data and the mathematical model. A tradeoff can be observed: if the mathematical assumptions are relaxed, more information about the system is required and vice versa. Furthermore, we discuss the limitations of the presented approach in the following.

Availability of Data: Service effect specifications have to be available for all services provided by a component. They have to be enriched with execution times, transition probabilities, loop iterations etc., and this information has to be specified by the component developer without knowing the usage context and deployment environment of the component. For the component developer, this can be a hard task that needs to be supported by tools guiding the estimation of resource demands or measuring the required data for existing components.

If a parameter influences the performance of a service, subdomains for its input values have to be identified by the component developer. This can be done by looking at boolean expressions of branches and loops that depend on the parameter (or one of the parameters that was derived from it). For example, the expression $(x < 5)$ implies a partitioning of the values of x into two subdomains: $x > 5$ and $x \leqslant 5$. For both subdomains, the system assembler can specify probabilities that are mapped to the branching probabilities according to the expected usage profile. As for the creation of SEFFs, it is the hope that the subdomains of parameters can be derived from a component's source code in a semi-automatic way.

Mathematical Assumptions: The stochastic regular expressions used in our prediction model are based on Markov chains. Therefore, some of the assumptions of Markov chains are inherited. The Markov property (the probability of going from state i to state j in the next step is independent of the path to state i) is present in our model, but has been weakened for loops. We explicitly model (arbitrary) PMFs for the number of loop iteration. Therefore, our prediction model is not bound to a geometrical distribution on the number of loop iterations like classical Markov models (also see [6]).

Branching probabilities are modelled in dependence on a service's input parameters. Thus, we still assume that the past history of the service's execution does not influence the branching probabilities. However, we allow parametrising these probabilities by characterisation of parameters of a service, thus enabling more realistic predictions for different usage contexts.

Many analytical performance prediction approaches assume that execution times are exponentially distributed, which significantly eases the analysis. However, the measurements of our case study in section 5 show that, often times, execution times are not exponentially distributed. For this reason, we used arbitrary distribution functions which reflect the actual system behaviour more accurately.

However, it is assumed that execution times are stochastically independent. This is a result of the convolution used to combine the execution times of sequential services. When convolving two PMFs, the result reflects all possible combinations of execution times. In reality, the execution times of sequential services might be dependent. For example, if the execution of one service is slow due to a high system load, the execution of another service will be slow as well. Such a dependency is not reflected by our model.

Further Limitations: Our approach is a first step and still embodies some limitations that shall be adressed in future research. The *scalability* of the approach is still unknown, as up to now we have not analysed a large-scale industrial size software architecture.

The *parameter modelling* is limited to primitive and composite data types, and parameters like streams or pointer are not supported.

So far, only one user request is modelled in the system at the same time, thus, *contention* for resources by concurrent requests is neglected in this approach. However, the results of our analyses can possibly used as input parameter for performance models such as queueing networks, which support contention analysis. Moreover, we do not support modelling components that start *threads* during the execution of their services, as we cannot analyse forks and joins in the control flow.

5 Case study

In the following we report on a case study to validate to applicability of our approach. The performance of a component-based on-line store for music files (WebAudioStore) is analysed. Parameters influence resource usage and inter-component control flow in this application, so the store is well suited to be modelled with our approach. Simplifying the analysed architecture aids in understanding the case study. However, the considered case is exemplary. Many similar cases could occur in an industrial sized architecture, whose analyses would be support by our method as well.

The architecture of the WebAudioStore has been modelled and implemented, so that measurements based on the implementation and predictions based on the specification can be compared. In this case, the performance prediction aims at supporting a design decision regarding an architectural alternative. The aim of the case study is to validate the applicability and usefulness of the proposed prediction model. Thus, the following questions have to be answered:

1. Does the prediction model favour the design with the lowest response time and, thus, support the right design decision?
2. How much do the computations based on component specifcations deviate from measurements based on an implementation?

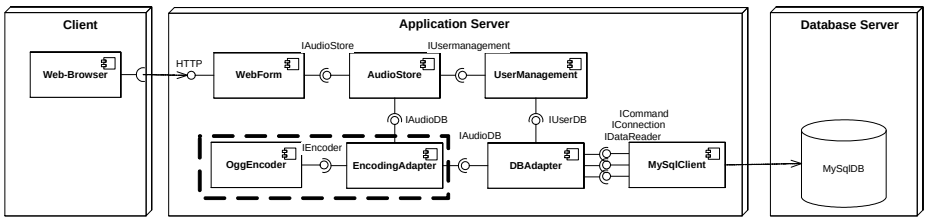


Fig. 3. WebAudioStore Architecture

5.1 Original Architecture

The simplified architecture of the WebAudioStore can be found in Fig. 3. Note that the components within the dashed box indicate an extension described in Section 5.2. Clients can buy and sell music files in the store via a web interface. To sell files, MP3-files can be uploaded to the store. It is possible to upload multiple files, so that complete albums can be offered. The files are stored in a MySQL database located on a different server than the application. Clients connect to the store using DSL lines (128 KBit/s Upload), the application server is connected to the database server with a dedicated line with a throughput of 512 KBit/s.

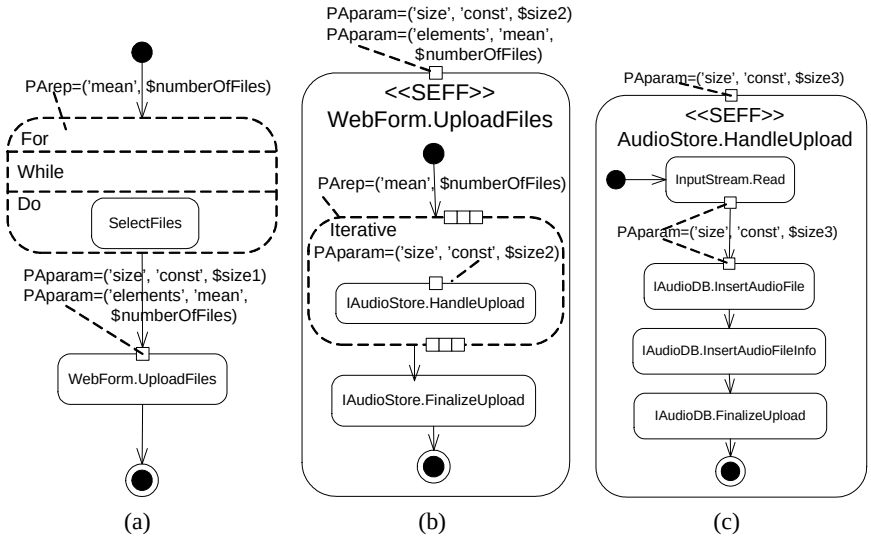


Fig. 4. Scenario for the Use Case "Upload Files"

Fig. 4(a) shows the usage scenario for uploading files to the store. Note that only the parameter dependencies are included in the illustration. Additional specifications necessary for the performance prediction like the service's resource demand are omitted in the illustration to allow an easier understanding. Users select several files from their hard drives and click the upload button afterwards, which initiates a service of the `WebForm` component. This is the performance critical service, since the files are copied to the database during this action.

Its SEFF (see Fig. 4(b)) indicates that the service `HandleUpload` of the component `AudioStore` is called as often as the number of files selected by the user. Thus, the inter-component control flow is influenced by a parameter provided by the user. The service `HandleUpload` (see Fig. 4(c)) calls services of the component `DBAdapter` (via the interface `IAudioDB`), which transmits the files to the database server by executing SQL queries. The size of the files influences the response time of this scenario.

The system architect can take these specifications provided by the component developers and instantiate the included variables with data from the usage scenario. In the scenario considered here, users usually upload eight to twelve MP3-files with a size of 3.5 to 4.5 MBytes. These files are encoded with a bit rate of 192Kbps.

The response times for this scenario are too slow and shall be improved transparently for the clients, so that they can use the store as usual.

5.2 Design Alternative: Compression

It is suggested to reduce the response time of the "UploadFiles" use case by applying the Fast Path performance pattern [15]. Thus, an additional compression component interface) is put between the `DBAdapter` and the `AudioStore` (dashed box in Fig. 3). By reducing the size of the uploaded audio files, the time for the network transfer between application server and database server is reduced. For the compression, a component using the OGG Vorbis encoder (component `OggEncoder`) shall be used that reduces the file sizes by one third by converting the MP3-files with a bitrate 192Kbps to OGG-files with a bitrate of 128Kbps. It is included into the architecture using the adapter `EncodingAdapter` that implements the `IAudioDB` interface. Because the audio quality of OGG-files with lower bitrates is better than the one of MP3-files, there are no significant quality losses expected. However, re-encoding the MP3-files costs a certain amount of time. With a performance prediction, it is analysed whether the time saved by reducing the network traffic outweighs the time for the encoding.

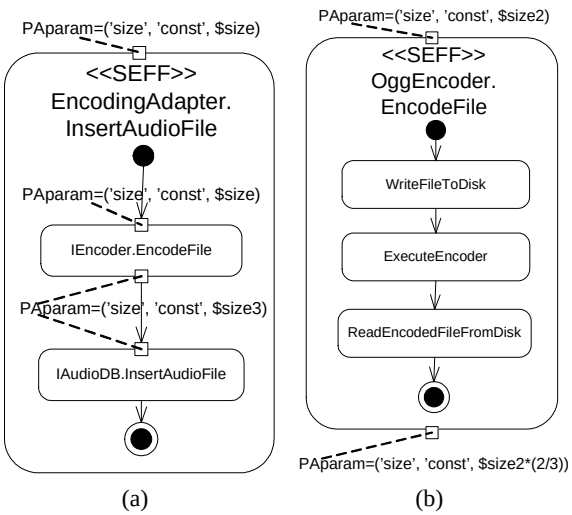


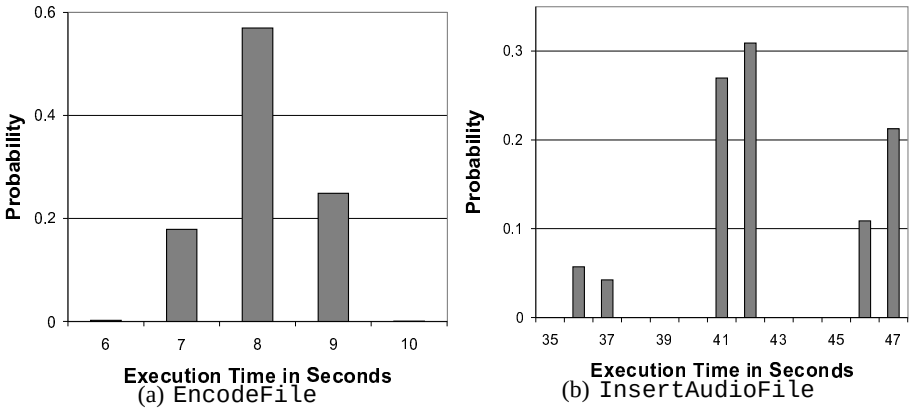
Fig. 5. SEFFs of the EncodingAdapter and OggEncoder

The component developer of the `OggEncoder` component has specified that the size of the output parameter of the `EncodeFile` service is $\frac{2}{3}$ of the input parameter's size (Fig. 5(b)).

5.3 Computations

Before answering the question which design alternative is rated best, we present how the computation process works and what input data was used for the example of the design alternative employing compression.

From the usage profile, it is known that the size of the input files is 3.5, 4, and 4.5 MB with a probability of 0.1, 0.6, and 0.3 respectively. The system assembler uses this information to estimate the execution times of the compression (Fig. 6(a)) and the transfer of the compressed file to the data base (Fig. 6(b)). For the estimation of the latter, the the compression rate of the `OggEncoder` has to be considered.



(a) `EncodeFile` (b) `InsertAudioFile`

Fig. 6. Probability mass functions used as input.

Both functions contain relatively few values and can easily obtained by either measurement or estimation. However, an integrated approach requires that both PMFs are derived automatically from the size of the input files and service specifications. For this paper, it is assumed that these values are delivered by the system assembler.

The encoding (`EncodeFile`) and transfer (`InsertAudioFile`) to the database are executed sequentially as shown in the SEFF of service `EncodingAdapter.InsertAudioFile` (Fig. 5(a)). To compute its execution time, the convolution of both PMFs is computed.

5.4 Results

Fig. 7 shows its result compared to the actual measurements. Even though the predictions match the measurements pretty well, they look a little bit "blurred". This is a result

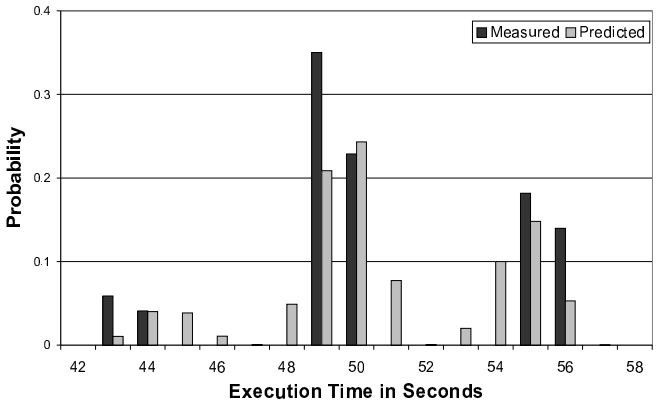


Fig. 7. Execution time of method `InsertAudioFile` of the `EncodingAdapter`.

of the convolution that computes all possible combinations of its input functions and, therefore, assumes their independence. This assumption does not hold in this case: If the file is large, both compression and transfer to the database will consume more time. To achieve more accurate results, this dependency needs to be reflected in the model.

Knowing the execution time of the `EncodingAdapter.InsertAudioFile` the execution time of the service `HandleUpload` of the `WebAudioStore` component is set to the same PMF, since the execution times of `FinalizeUpload` and `InsertAudioFile` are below one second and are thus set to zero.

As the last step, the execution time of the service `UploadFiles` of the `WebForm` component is determined using the computed values as input. The execution time of `FinalizeUpload` is assumed to be zero. The usage profile contains information on the value distribution of parameter `numberOfFiles`. This is used for the computation of the loop execution time. It is known, that eight to twelve files are uploaded by the users with a probability of 0.1, 0.1, 0.2, 0.4, and 0.2 respectively. This directly influences the number of loop iterations as expressed by the `PArep` tag in Fig. 4(b). Fig. 8 shows the resulting prediction in comparison to the measurements. The curve is not an exact fit, but represents its structure pretty well. For the original architecture, the predictions are closer to the measurements (Fig. 8). This is due to the fact that the error introduced by the assumption of independence in the predictions does not play a role here. Only the execution time of the service `DBAdapter.InsertAudioFile` is influenced by the file size of the uploaded files.

Fig. 9 depicts what is predicted if the information on the parameters is neglected and a common Markovian modelling is applied (the underlying problem is also described in [6]). Instead of executing the loop in SEFF `WebForm.UploadFiles` (Fig 4(b)) eight to twelve times as specified by the input parameter `numberOfFiles`, a loop probability p was used to determine whether the loop is (re-)entered (with probability p) or left (probability $1-p$). Thus, the loop is never iterated with probability $1-p$, once with probability $p(1-p)$, twice with probability $p^2(1-p)$, and so on [7]. Thus, the

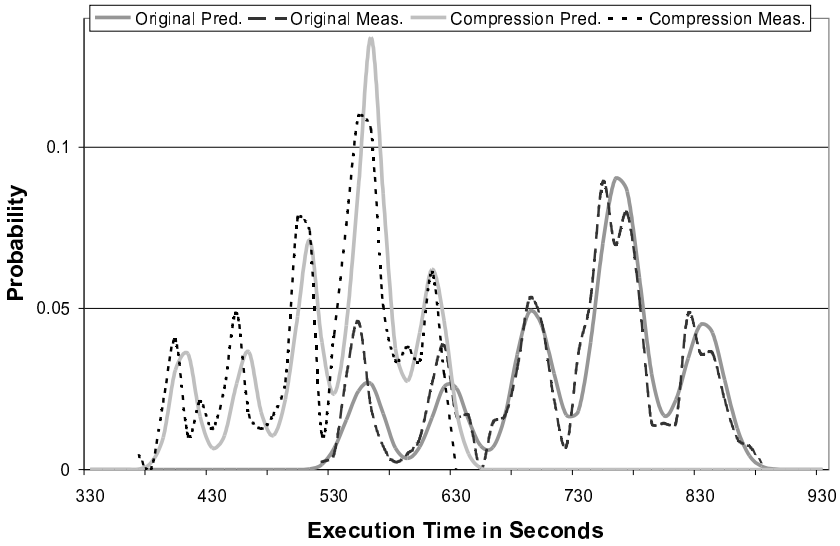


Fig. 8. Execution time of method `UploadFiles` of the `WebForm`.

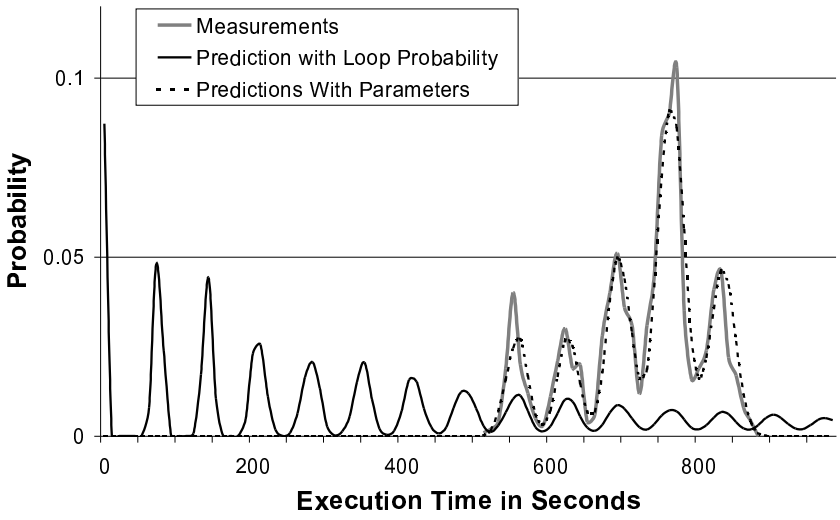


Fig. 9. Execution time of method `UploadFiles` of the `WebForm`.

number of loop iterations is geometrically distributed. This influence can be observed at the predicted execution time. The probability of not executing the loop is highest, after that the probability decreases and converges to zero. Obviously, the predicted curve does not match the measurements in any aspect. This shows that the Markov property for loops (the probability of re-iterating the loop does not change over time) does not hold in this case. This was to be expected and can be handled by the prediction model for loops used in our approach.

The results described above answer the questions asked in the beginning of this section. The prediction model favored the design alternative with compression, which was also the fastest during our measurements. Thus, the first question can be answered with "yes". The PMFs shown in Fig. 8 answer the second question in a non-formal way. To answer the question completely, a proper measure for the error of two PMFs describing execution times has to be found and applied to measured and predicted functions. However, a detailed analysis of the error made by the predictions is beyond the scope of this paper.

6 Related Work

The SPE methodology [16] was one of the first approaches to analyse the performance of a software system during early development stages. A survey on model-based performance prediction approaches is provided in [1]. Specifically for component-based performance predictions, there is a survey on approaches related to the one presented here in [2].

The CB-SPE approach by Bertolino et. al. [4] uses sequence diagrams and queueing networks to analyse the performance of component-based software systems. For each service, the performance is specified in dependency of so-called environment parameters like CPU time or network bandwidth. There is no characterisation of parameters passed by users to a service in this approach.

Hamlet et. al. [8] presented an approach for the performance analysis of component-based systems that relies on measurements. In this more theoretical approach, components compute single functions and their input space is divided into subdomains by profiling them. Subdomains are only created for the values of parameters, whereas in our approach we also allow to specify subdomains over the number of elements in a collection or the byte size of a parameter.

Bondarev et. al. [5] explicitly model input parameters of software components and make performance predictions. However, there is no probabilistic characterisation of parameter values in this approach, as it is assumed that a fixed parameter assignment can be identified in a certain scenario, which may be realistic for the embedded systems the approach is aiming at.

Sitaraman et. al. [14] also aim at performance predictions incorporating parameter values. In their approach, parameters are characterised using a modified form of the Big-O Notation. However, it is not shown how this characterisation can be transformed into timing values.

7 Conclusions and Future Work

An approach including the dependencies between component service parameters and performance has been presented in this paper. Service effect specifications modelling external calls of a component service were extended to include parameter dependencies using a notation based on the UML SPT profile. The case study of an component-based online shop showed that the method can support design decisions during early development stages. Parameter dependent performance specification can lead to more refined and accurate predictions. The approach is especially suited to model systems with extensive data flow, because the size of data packets transferred between components can be included into the predictions.

However, there are several pointers for future work. Modelling concurrency (e.g., multiple threads) is not supported by the method presented here and will be included in the future. Parameter dependencies can also be expressed as OCL constraints, thus existing OCL checkers could be used to validate the syntax. We will explore this direction in the future. More complex parameters like streams or pointers can not be modelled. So far, all necessary specifications have to be created by component developers manually. Thus, code analysis techniques shall be used in the future to generate parts of these specifications from source code of existing components semi-automatically.

Acknowledgements: We would like to thank Ralf Reussner and Viktoria Firus for their ideas and the fruitful discussions.

References

1. S. Balsamo, A. DiMarco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, May 2004.
2. S. Becker, L. Grunske, R. Mirandola, and S. 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*, number To Appear in LNCS. Springer, 2005.
3. S. Becker and R. Reussner. The Impact of Software Component Adaptation on Quality of Service Properties. *L'objet*, 12(1):105–125, 2006.
4. A. Bertolino and R. Mirandola. CB-SPE Tool: Putting component-based performance engineering into practice. In I. Crnkovic, J. A. Stafford, H. W. Schmidt, and K. C. Wallnau, editors, *CBSE2004*, volume 3054 of *LNCS*, pages 233–248. Springer, 2004.
5. E. Bondarev, P. de With, M. Chaudron, and J. Musken. Modelling of input-parameter dependency for performance predictions of component-based embedded systems. In *Proceedings of the 31th EUROMICRO Conference (EUROMICRO'05)*, 2005.
6. K. Doerner and W. Gutjahr. Representation and optimization of software usage models with non-markovian state transitions. *Information & Software Technology*, 42(12):873–887, 2000.
7. V. Firus, S. Becker, and J. Happe. Parametric performance contracts for QML-specified software components. In *Proceedings of FESCA2005*, ENTCS, pages 64–79, 2005.
8. D. Hamlet, D. Mason, and D. Woit. *Properties of Software Systems Synthesized from Components*, volume 1 of *Series on Component-Based Software Development*, chapter Component-Based Software Development: Case Studies, pages 129–159. World Scientific Publishing Company, March 2004.

9. H. Koziolek and J. Happe. A quality of service driven development process model for component-based software systems. In *Proceedings of the 9th International Symposium on Component Based Software Engineering (CBSE2006)*, 2006.
10. H. Koziolek and V.Firus. Parametric performance contracts: Non-markovian loop modelling and an experimental evaluation. In *Proceedings of FESCA2006*, ENTCS, 2006.
11. M. Marzolla. *Simulation-Based Performance Modeling of UML Software Architectures*. PhD thesis, Universit'a Ca Foscari di Venezia, 2004.
12. Object Management Group OMG. UML Profile for Schedulability, Performance and Time. http://www.omg.org/cgi-bin/doc?formal/2005-01-02, 2005.
13. R. Reussner, S. Becker, and V. Firus. Component composition with parametric contracts. In *Tagungsband der Net.ObjectDays 2004*, pages 155–169, 2004.
14. M. Sitaraman, G. Kuczycki, J. Krone, W. F. Ogden, and A.L.N. Reddy. Performance specification of software components. In *Proc. of SSR '01*, 2001.
15. C. U. Smith. *Performance Solutions: A Practical Guide To Creating Responsive, Scalable Software*. Addison-Wesley, 2002.
16. C.U. Smith. *Performance Engineering of Software Systems*. Addision-Wesley, 1990.
17. X. Wu and M. Woodside. Performance modeling from software components. *SIGSOFT Softw. Eng. Notes*, 29(1):290–301, 2004.