Model-Based Performance Prediction with the Palladio Component Model

Steffen Becker
IPD, University of Karlsruhe
76131 Karlsruhe, Germany
sbecker@ipd.uka.de

Heiko Koziolek*
Graduate School TrustSoft
University of Oldenburg
26111 Oldenburg, Germany
heiko.koziolek@trustsoft.uni-oldenburg.de

Ralf Reussner
IPD, University of Karlsruhe
76131 Karlsruhe, Germany
reussner@ipd.uka.de

ABSTRACT

One aim of component-based software engineering (CBSE) is to enable the prediction of extra-functional properties, such as performance and reliability, utilising a well-defined composition theory. Nowadays, such theories and their accompanying prediction methods are still in a maturation stage. Several factors influencing extra-functional properties need additional research to be understood. A special problem in CBSE stems from its specific development process: Software components should be specified and implemented independent from their later context to enable reuse. Thus, extra-functional properties of components need to be specified in a parametric way to take different influence factors like the hardware platform or the usage profile into account. In our approach, we use the Palladio Component Model to specify component-based software architectures in a parametric way. This model offers direct support of the CBSE development process by dividing the model creation among the developer roles. In this paper, we present our model and a simulation tool based on it, which is capable of making performance predictions. Within a case study, we show that the resulting prediction accuracy can be sufficient to support the evaluation of architectural design decisions.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; C.4 [**Performance of Systems**]; I.6.5 [**Simulation and Modelling**]: Model Development

Keywords

Component-Based Software Engineering, Software Architecture, Performance Prediction

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

1. INTRODUCTION

In CBSE, a central idea is to build complex software systems by assembling basic components. The initial goal of CBSE was to increase the level of reuse. However, composite structures may also increase the predictability of the system during early design stages, because models of individual components can be certified and then be composed, enabling system architects to reason on the composed structure. This is important for functional properties, but also for extra-functional properties like performance (i.e., response time, throughput, resource utilisation) and reliability (i.e., mean time to failure, probability of failure on demand).

Prediction methods for performance and reliability of general software systems are still limited and seldomly used in industry [2, 4]. Especially for component-based systems further challenges arise. Opposed to object-oriented system development and performance prediction [21], where developers design and implement the whole system, several independent developer roles are involved in the creation of a component-based software system. Component developers produce components that are assembled by system architects and deployed by system allocators. The diverse information needed for the prediction of extra-functional properties is thus spread among these developer roles.

Most existing methods for component-based performance prediction require system architects to model the system based on specifications of single components. Often, it is assumed that the system architect can provide missing information. This assumption is necessary because of today's incomplete component specifications. For example, in [5] system architects model the control flow through the component-based architecture, which is impossible if components are black boxes and the dependencies between provided and required interfaces are unknown. Thus, a special component specification is needed.

Other approaches neglect factors affecting the perceived performance of a software component like influences by external services [20, 9], changing resource environments [12, 17, 6], or different input parameters [5]. However, for accurate predictions, these dependencies have to be made explicit in component specifications.

With the Palladio [1] Component Model, a meta-model al-

[1]Our component model is named after the Italian renaissance architect Andrea Palladio (1508-1580), who, in a certain way, tried to predict the aesthetic impact of his buildings in advance.

lowing the specification of performance-relevant information of a component-based architecture, we provide an initial attempt to address the identified problems. First, our model is designed with the explicit capability of dividing the model artefacts among the different roles involved in a CBSE development process. These modelling artefacts can be considered as domain specific modelling languages, which capture the information available to a specific developer role. Second, the model reflects that a component can be used in changing contexts with respect to the components it is connected to, the allocation of the component on resources, or different usage contexts. This is done by specifying parametric dependencies, which allow deferring context decisions like assembly or allocation.

For an initial validation, we have developed a tool capable of simulating instances of the Palladio Component Model to obtain performance metrics. We used this tool in a case study to simulate the performance of a component-based online shop. Comparing the simulation results with measurements made on an implementation of the architecture enabled estimating the accuracy of our simulations.

The contribution of this paper is a component meta-model based on a CBSE role concept which allows parametric specifications done by different developer roles in the CBSE development-process. Additionally, we present mathematical model concepts of our component model, which allow to use arbitrary stochastic distribution functions for these types of specifications. A case study which applies our simulation tool for instances of our meta-model to the implementation of the modelled architecture finally shows the expressiveness of our model. Assumptions we made in earlier work [15] have been weakened in this case study but still without significant loss of prediction accuracy.

This paper is structured as follows: In section 2, we briefly review related work. Section 3 introduces our CBSE role concept and provides details of the component meta-model. Examples of how the parametric dependencies can be specified and evaluated are given in section 4. Section 5 details on the developed simulation tool. Assumptions and limitations of our work are discussed in section 6. In section 7, a case study applying our simulation tool to a model instance is presented. Finally, we conclude our paper and outline future work.

2. RELATED WORK

The approach presented in this paper is related to performance meta-models, component models, usage modelling in performance prediction, and simulations.

Three recent *performance meta-models* are compared by Cortellessa in [7]: The performance domain model of the UML SPT profile [18], the Core Scenario Model from Woodside et al. [23], and the Software Performance Engineering (SPE) meta-model [21] are designed for general software systems. Another meta-model is KLAPER from Grassi et al. [11], which, like our model, is designed for component-based software systems. KLAPER reduces the complexity of other models with a unifying concept for resources and components. The Palladio Component Model reduces modelling complexity by providing different models for different CBSE developer roles.

Besides models designed specifically for the runtime performance prediction of a software system, several other *component models* have been proposed. Each model has its own special focus on a set of particular aspects depending on the respective analysis methods. Recent component models are often divided into two types: industrial- and research-oriented models. Industrial models (like EJB or COM) have been designed to support specific implementation tasks. They often lack the support of broad analysis capabilities w.r.t. extra-functional properties. Research oriented models (like SOFA) are often accompanied with a special analysis method for a set of system properties. A recent taxonomy of the models used today is presented in [16].

Other approaches have put emphasis on accurate *usage modelling* for performance predictions. Hamlet et al. [12] execute components and measure how they propagate requests in order to gain accurate performance predictions. In our approach, we require component developers to specify these propagations, because components are often not available for measurements when including them into predictions. Bondarev et al. [6] model cost functions depending on input parameters, but do not use stochastic characterisations of these parameters. Sitamaran et al. [20] use extended Big-O Notations to specify the performance of software components depending on input parameters. Bertolino et al. extend the SPE approach for component-based systems in [5], but do not model input parameters.

Simulation techniques are often used to evaluate performance models such as queueing networks, stochastic Petri nets, and stochastic process algebras. In a survey on model-based performance predictions techniques by Balsamo et al. [2], simulations models by [8] and [1] are described. The UML-PSI tool by Marzolla [3] derives an event-driven simulation from UML models, but is not specifically designed for component-based systems.

3. COMPONENT-BASED PERFORMANCE MODELLING

The Palladio Component Model is a meta-model for the description of component based software architectures. The model is designed with a special focus on the prediction of Quality-of-Service (QoS) attributes, especially performance and reliability. In the following, we give some details on our envisioned CBSE development process and the participating roles. Afterwards, we highlight some concepts of our meta-model omitting concepts not used in this paper.

3.1 CBSE Development Process

In the CBSE development process (see also [14]), we distinguish four types of developer roles involved in producing artefacts of a software system (see Figure 1).

Component developers specify and implement the components. The specification contains an abstract, parametric description of the component and its behaviour. *System architects* assemble components to build applications. For the prediction of extra-functional properties, they retrieve component specifications by component developers from a repository.

System deployers model the resource environment and afterwards the allocation of components from the assembly model to different resources of the resource environment. Business *domain experts*, who are familiar with the customers or users of the system, additionally provide usage models describing critical usage scenarios as well as typical parameter values.

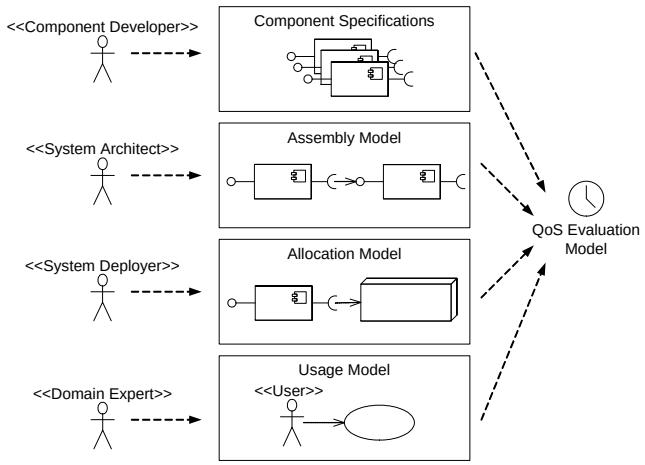

Figure 1: Process

The complete system model can be derived from the partial models specified by each developer role and then extra-functional properties can be predicted. Each developer role has a domain specific modelling language and only sees and alters the parts of the model in its responsibility. The introduced partial models are aligned with the reuse of the software artefacts.

3.2 Fundamental Concepts

Several concepts in the Palladio Component Model have counterparts in the UML2 meta-model. Hence, we keep the description of these concepts brief. We do not define a profile for UML2, because we want to avoid ambiguities in the UML2 meta-model and restrict the specification only to those constructs that can be handled by our evaluation method.

`Components` are generally specified via provided and required interfaces. An interface serves as contract between a client requiring a service and a server providing the service. Components implement services specified in their provided interfaces using services specified in their required interfaces.

In its most basic form, an `Interface` consists of a list of service `Signatures`. A signature has a name, a sorted list of parameters, a return type and an unsorted list of exceptions it might raise during its execution. This is similar to the Corba Interface Definition Language (IDL). Interfaces are first-class entities in the Palladio Component Model and themselves neither providing nor requiring. Only their relations to components define their roles. We call this relation `Provided-` or `Required Role`, respectively.

Components and their roles can be connected to build an `Assembly` using assembly connectors. An assembly connector connects a required role of a component with a provided role of another component. This means that any call emitted by the component requiring a service of the required role is directed to the connected component providing that service. For connectors it is important that the required and provided interfaces match, e.g., that the service is provided as expected by the requiring component. As mentioned earlier, the assembly model is specified by the system architect in a domain specific modelling language referring to specifications of individual components from component developers.

3.3 Service Effect Specification

To each provided service of a component, component developers can add a so-called `ServiceEffectSpecification` (SEFF), which describes how the provided service calls the required services of the component. In former approaches (e.g., [19]), SEFFs have been described as automata modelling the order of calls to required services thus being an abstraction of the control flow through the component.

For performance analysis, the mere sequence of external calls is not sufficient. Thus, we extend SEFFs to so-called `ResourceDemandingSEFF`s. Besides the sequence of called required services, a `ResourceDemandingSEFF` contains resource usage, transition probabilities, loop iteration numbers, and parameter dependencies to allow accurate performance predictions. Its meta-model will be described in the following. It can be considered as a domain specific modelling language for the component developer to specify performance related information for a component service. For clarity, we spread the description over multiple figures. Examples for `ResourceDemandingSEFF`s follow in figures 8-13.

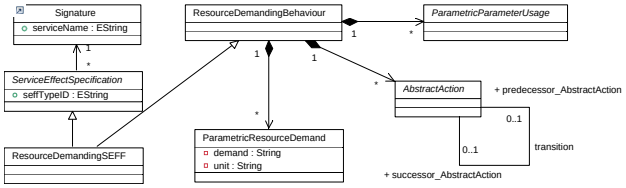

Figure 2: Behaviour

Figure 2 shows that a `ResourceDemandingSEFF` extends a `ServiceEffectSpecification`, which itself references the `Signature` of a service, and a `ResourceDemandingBehaviour`. A `ResourceDemandingBehaviour` consists of a number of `AbstractActions`, a number of `ParametricResourceDemands`, and a number of `ParametricParameterUsages`.

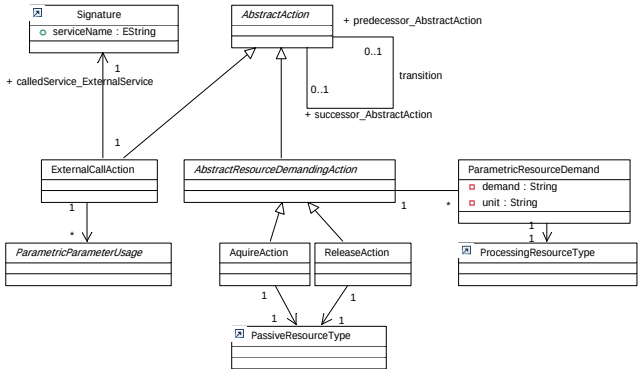

Figure 3: Abstract Actions

An `AbstractAction` (Figure 3) can be specialised to be an `AbstractResourceDemandingAction` or an `ExternalCallAction` to a required service. `ResourceDemandingActions` can place loads on the resources, which the component is using (e.g., CPU, harddisk, network connection, etc.). Demands can be specified as distribution functions, additionally their unit (e.g., CPU operations, harddisk accesses, etc.) has to be specified. Because the actual resources used by

the component are not known during component specification, the component developer specifies the resource demand only for abstract resource types (in this case `ProcessingResourceTypes`) and not for concrete resource instances.

Passive resources, such as threads or semaphors, have to be acquired before using them, and released afterwards. This can be modelled with the `AcquireActions` and `ReleaseActions` that are associated with a `PassiveResourceType`.

The resource usage generated by executing a required service with an `ExternalCallAction` has to be specified in the SEFF of that required service, and is not part of the SEFF of the component requiring it. An `ExternalCallAction` is always associated with the signature of another service. Parameters can be passed with `ExternalCallActions`. If they influence QoS properties, they should be characterised by a `ParametricParameterUsage` (see section 4.4 for further details).

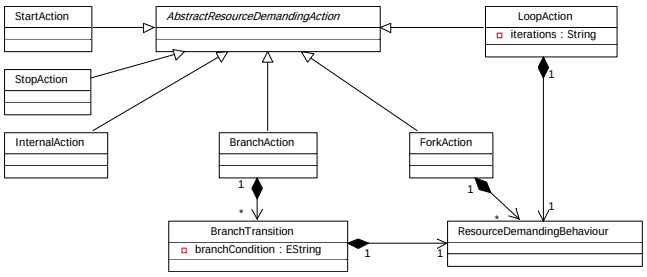

Figure 4: Actions

Figure 4 shows different specialisations of `AbstractResourceDemandingAction`. An `InternalAction` models component-internal computations of a service possibly combining a number of operations in a single model entity. The algorithms executed internally by the component are thus not visible in the SEFF to preserve the component blackbox principle.

Furthermore, each `ResourceDemandingBehaviour` has one `StartAction` with only successors and possibly several `StopActions` with only predecessors. `AbstractActions` are arranged into sequences by associating each one with its predecessor and successor. Common control flow primitives, such as branch, loop, and fork can also be used to connect `AbstractActions`.

`LoopAction`, `BranchTransition`, and `ForkAction` contain inner `ResourceDemandingBehaviours`, which again can be modelled with `ResourceDemandingActions`. Modelling inner behaviours reduces the amount of ambiguities in the specification and eases the later analysis. For example, merging formerly branched control flow paths is well-defined with inner behaviours.

A `BranchAction` consists of a number of `BranchTransitions`, which are attributed with branch probabilities.

A `LoopAction` is attributed with the number of iterations. Control flow cycles always have to be modelled explicitly with `LoopActions`, thus an `AbstractAction` is not allowed to have another action as one of its predecessors *and* at the same time one of its successors. This way, it is disallowed to specify loops with a probability of entering the loop and a probability of exiting the loop as it is done for example in Markov models. Loops in Markov models are restricted to a geometrically distributed number of iterations, whereas our evaluation model supports a generally distributed or even fixed number of iterations, which allows analysing many loops more realistically (for more details on loop modelling, see [13]).

So far, we require component developers to model SEFFs manually by examining the code of their components. In the future, we aim at developing static code analysis techniques and tools assisting in generating these specifications semi-automatically from component source code.

3.4 Usage Model

As the usage of a component-based system is relevant for QoS analyses, the Palladio Component Model contains an usage model, which allows describing workloads and usage scenarios. It is a domain specific modelling language intended for business domain experts, who are able to describe the expected user behaviour of the system. System architects may also construct usage models from requirements documents.

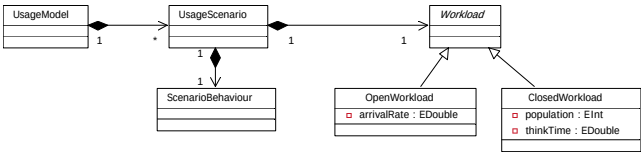

Figure 5: Usage Model

The `UsageModel` (Figure 5) consists of a number of `UsageScenarios`, which in turn consist of one `ScenarioBehaviour` and one `Workload` meaning that the `ScenarioBehaviour` is executed with the respective `Workload`. `Workloads` describe the usage intensity of the system. They can be `OpenWorkloads`, modelling users that enter with a given arrival rate and exit the system after they have executed their scenario. Or they can be `ClosedWorkloads`, modelling a fixed number of users (population), that enter the system, execute their scenario and then re-enter the system after a given think time. This resembles the common modelling of workloads used in performance models like queueing networks.

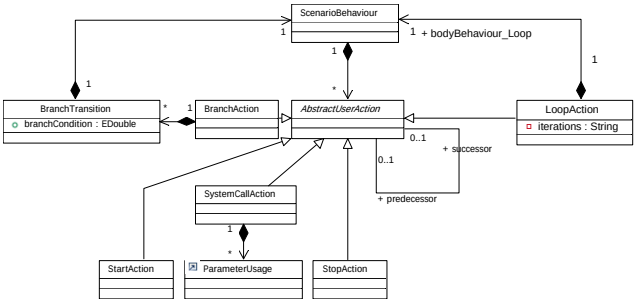

Figure 6: Scenario Behaviour

A `ScenarioBehaviour` (Figure 6) consists of a sequence of `AbstractUserActions` (notice the predecessor, successor association). These can be specialised into control flow primitives (`BranchAction` and `LoopAction`) or `SystemCallActions`, which are calls to component interfaces directly accessible by users (on the entry level of the architecture). Forks

in the user control flow are not possible, as it is assumed that one user can only execute a single task at a time. A `ScenarioBehaviour` does not contain resource usage, which only can be generated by components.

`SystemCallAction`s contain a number of `ParameterUsage`s to characterise actual parameters.

3.5 Parameter Model

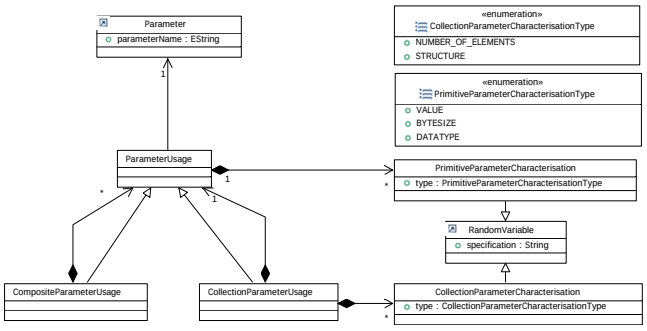

Figure 7: Parameter Usage

In addition to formal parameters (`Parameter`), actual parameters can be characterised with `ParameterUsage`s (Figure 7). These characterisations have been modelled especially for QoS analyses. `ParameterUsages` of primitive parameters (such as int, float, char, boolean, etc.) can be characterised by their value, bytesize and type.

`CollectionParameterUsages` (for arrays, lists, sets, trees, hash maps, etc.) are `ParameterUsages` themselves and can additionally be characterised by the number of inner elements and their structure (e.g., sorted, unsorted etc.). They contain `ParameterUsages` to abstractly characterise their inner elements.

In a `CompositeParameterUsage` different primitive and collection parameters can be grouped together with an inner `ParameterUsage` for each of the grouped parameters. With this abstraction, more complex parameters (such as records, objects, etc.) can be modelled.

The different attributes like value or bytesize can be specified as random variables, allowing for example to specify a distribution function for the bytesizes of inner elements of a collection.

3.6 Resource Model

The Palladio Component Model differentiates between three types of resources: active resources, passive resources, and linking resources. Active resources process jobs on their own (e.g., a CPU processing jobs or a hard disk processing read and write requests). Opposed to this, passive resources do not process jobs on their own. Instead, their possession is important as they are limited. For example, if there is only one database connection it can only be used by one component and all other components in need for the same connection have to wait. Linking resources enable modelling of communication. It can be any kind of network connection but also some abstraction of it, like a RPC call.

Active and passive resources are bundled in resource containers. A resource container is similar to a UML2 deployment node. Resource containers are connected by linking resources. The complete resource model is specified by system deployers, who also assign components to specific resources. In the future, a more refined resource model could be designed according to the General Resource Model (GRM) of the UML SPT profile [18].

4. PARAMETRIC DEPENDENCIES

The performance of a software component is influenced by its *usage* [12]. The resource demand may vary depending on input parameters (e.g., uploading larger files with a component service produces a higher demand on hard disk and network). Different required services can be called as a result of different inputs, thus the branch probabilities in the SEFF are most often linked to the usage profile (e.g., required service `A` is called if some integer parameter is larger than zero, otherwise service `B` is called). Furthermore, the parameters passed to required services (forming a usage model for the required component) may also depend on a service's own input parameters.

The central dilemma of the component developer is that during component specification it is unknown how the component will be used by third parties. Thus, in case of varying resource demands or branch probabilities depending on user inputs, the component developer cannot specify fixed values. However, to help the system architect in QoS predictions, the component developer can specify the *dependencies* between input parameters and resource demands, branch probabilities, or loop iteration numbers in SEFFs. If an usage model of the component has been specified by business domain experts or if the usage of the component by other components is known, the actual resource demands and branch probabilities can be determined by the system architect by solving the dependencies.

In the Palladio Component Model, we use *random variables* to express resource demands or numbers of loop iterations. Mathematically, a random variable is defined as a measurable function from a probability space to some measurable space. More detailed, a random variable is a function $X : \Omega \rightarrow \mathbb{R}$ with Ω being the set of observable events and $\mathbb{R}$ being the set associated to the measurable space. Observable events in the context of software models can be for example response times of a service call, the execution of a branch, the number of loop iterations, or abstractions of the parameters, like their actual size or type.

A random variable X is usually characterised by stochastical means. Besides statistical characterisations, like mean or standard deviation, a more detailed description is the probability distribution. A probability distribution yields the probability of X taking a certain value. It is often abbreviated by $P(X = t)$. For discrete random variables, it can be specified by a probability mass function (PMF), as used in our component model. The event spaces Ω we support include integer values $\mathbb{N}$, real values $\mathbb{R}$, boolean values and enumeration types (like "sorted" and "unsorted").

Additionally, it is often necessary to build new random variables using other random variables and mathematical expressions. For example, to denote that the response time is 5 times slower, we would like to simply multiply a random variable for a response time by 5 and assign the result to a new random variable. For this reason, our specification language supports some basic mathematical operations ($*$,$-$,$+$,$/$,...) as well as some logical operations for boolean type expressions ($==$,$>$,$<$,`and`,`or`,...).

We use the introduced random variables in the following

to provide several examples for specifying dependencies between input parameters and QoS-related specifications. We also use random variables in the case study to characterise the parameters of the calls issued to the system.

4.1 Branch Conditions

In Figure 8, the `ResourceDemandingSEFF` of the service HandleShipping from an online-store component is depicted. It has been specified by a component developer in a parametrised form. The service calls required services shipping a customer's order with different charges depending on its costs, which it gets passed as an input parameter. If the order's total amount is below 100 Euros, the service calls a service preparing a shipment with full charges (ShipFullCharges). If the costs are between 100 and 200 Euros, the online store grants a discount, so ShipReducedCharges is called. Orders priced more than 200 Euros are shipped for free with the ShipWithoutCharges service.

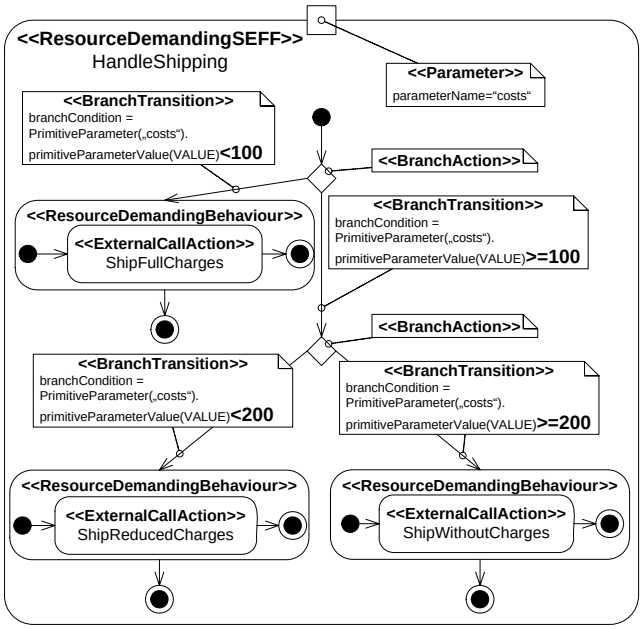

Figure 8: Branch Condition Example

The `ResourceDemandingSEFF` in Figure 8 is an abstract representation of the control flow through the component. Internal computations as well as resource demands of the service have been abstracted away, because they are not relevant for QoS analysis in this case.

Once a domain expert specifies the value of the parameter `costs`, it can be derived which of the services will be called. In this example, the domain expert has specified the value as a PMF (Table 1) according to a customer analysis. The PMF is used in order represent a whole group of customers using the service. It has been specified within an `UsageModel`, which is not illustrated here for brevity. In this case, the sample space Ω of the PMF consists of integer values $\mathbb{N}$ representing the costs of an order. Note, that the specification of the domain expert only refers to parameters visible at the service interface. The usage specifications can be made without referring to internals of the component thus preserving the black box principle.

Costs (Euro)	Probability
0-49	0.35
50-99	0.25
100-149	0.20
150-199	0.15
200-	0.05

Table 1: Probability of costs; specified by Domain Expert

To determine the branch probabilities which are required for the QoS analysis, random variables associated with the branches have to be evaluated. For the first branch node, the left and right branch conditions are denoted as A and B, where A is the event that costs are below 100 Euro and B the event that costs are larger than 100 Euros. With the usage profile by the domain expert, their probabilities of becoming true can be computed as: $P(A) = 0.35 + 0.25 = 0.6$ and $P(B) = 0.20 + 0.15 + 0.05 = 0.4$.

After any branching event X has occured, the sample space Ω on subsequent nodes is restricted to $\Omega' = \Omega \cap X$. This has to be considered by the following evaluations. In the example, this situation occurs at the second branch node. The left and right branch conditions are denoted as C and D, where C is the event that the costs are below 200 Euros and D the event that the costs are larger than 200 Euros. When computing the probabilities, it has to be taken into account, that the sample space has been restricted by B. Thus, a new sample space $\Omega' = \Omega \cap B$ has been created. Using the usage profile from the domain expert, the conditional probabilities are computed as: $P(C|\Omega') = P(C \cap \Omega')/P(\Omega') = 0.35/0.4 = 0.875$ and $P(D|\Omega') = P(D \cap \Omega')/P(\Omega') = 0.05/0.4 = 0.125$.

4.2 Loop Iterations

In the Palladio Component Model, it is possible to assign a number of iterations to a loop. This can be done in a parameterisable form, as illustrated by the following example. Figure 9 shows the `ResourceDemandingSEFF` of the service UploadFiles. It gets an array of files as input parameter and calls the external service HandleUpload within a loop for each file. As an example, the component shall be used in a new architecture for an online music repository. Users shall upload music albums via the service, which are then stored one by one in a database that is connected to the service HandleUpload.

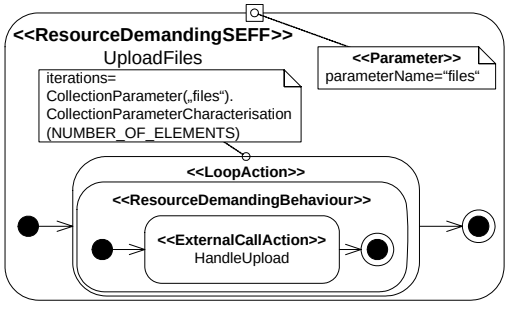

Figure 9: Loop Example

A domain expert has analysed user behaviour and found

Number of Files	Probability
8	0.1
9	0.1
10	0.2
11	0.4
12	0.2

Table 2: Probability for number of files; specified by Domain Expert

that users usually upload albums with 8-12 music files. Thus, a PMF for the number of files in the input parameter `files` has been specified (Table 2).

With the specified dependency to the number of elements in the input collection, the probability distribution of random variable X_{iter} for the number of loop iterations in the `ResourceDemandingBehaviour` can be determined. The required service HandleUpload will be called 8 times (probability 0.1), 9 times (0.1), 10 times (0.2), and so on (see Table 2). If the dependency had not been specified, it would not have been known from the interfaces how often the required service would have been called. Thus, with the specified PMF, a more refined prediction can be made for varying usage contexts.

4.3 Parametric Resource Demand

Besides branch conditions and loop iterations, resource demands can be specified in a parameterised form. In many cases, this is the most influencing factor for varying response times. In Figure 10, the component service ProcessFile receives an input parameter, which is processed internally. The component developer has specified that the resource demand of this service depends on the size of the input file, in particular 3 CPU operations are executed for each byte of the file.

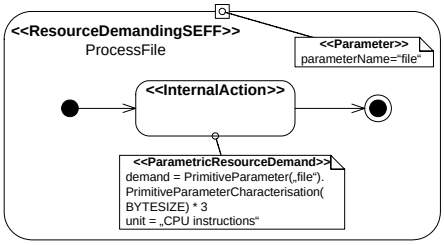

Figure 10: Resource Demand Example

Because of varying file sizes due to varying usages, the domain expert has specified a PMF for the size of the input file. After analysing a large number of usage traces from a similar system, a fine grain distribution function could be specified, which is shown in Figure 11. Note, that in this example the size of the file is the only attribute relevant for the QoS analysis. Other attributes of the file parameter, such as the value or the type of the file, are irrelevant in this case and need not to be specified. This is an example of abstracting unnecessary details from the model and in many cases such abstractions model reality good enough to make sufficiently accurate predictions.

To compute the actual resource demand on the CPU from the specification in Figure 10, the underlying PMF can be

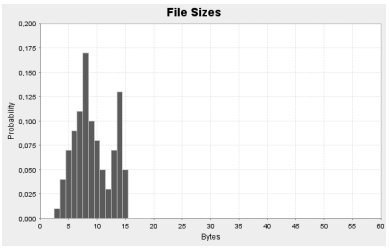

Figure 11: File Sizes (PMF), from Domain Expert

obtained by multiplying the PMF for the file sizes by a factor of 3, thus streching the PMF as depicted in Figure 12. Instead of file sizes, the PMF now denotes the probabilities for the number of CPU operations, which are executed for the given usage context. Once the actual CPU is known from the allocation model, on which the component is deployed, the time for executing service ProcessFile can be computed after specifying the execution time for a single CPU operation.

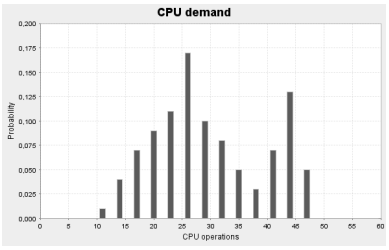

Figure 12: CPU operations (PMF), computed

4.4 Parametric Parameter Usage

When calling required services, component services may pass parameters. In many cases, these parameters can be fixed in the implementation. However, sometimes, input parameters for required service calls actually depend on the provided service's own input parameters. In the Palladio Component Model, such a dependency can be expressed by attaching a `ParametricParameterUsage` to an `ExternalCallAction`.

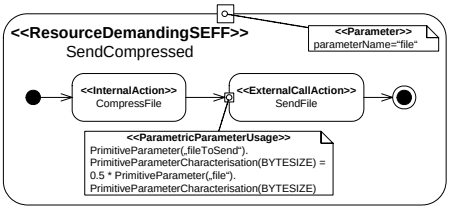

Figure 13: Parametric Parameter Usage Example

As an example, in Figure 13, the `ResourceDemandingSEFF` of the service SendCompressed is shown. It receives a `file` as an input parameter, compresses it using a ZIP algorithm, and then passes it to another component by calling the service SendFile. The component developer has specified that the compression reduces the size of the input files by 50%.

If a domain expert specifies the input file size, or if it has been specified in the `ParametricParameterUsage` of another component calling this component, the file size for the input parameter of the `ExternalCallAction` can be determined. If the file size has been specified as a PMF (like in the previous example), its domain values have to be multiplied by a factor of 0.5, thus contracting the PMF.

5. SIMULATION FRAMEWORK

To evaluate the model concepts introduced in the previous sections, we have built a prototypical simulation tool. This tool takes an instance of the Palladio Component Model and builds a simulation in order to get the response times of the specified workloads. In so doing, we validate two things. First, if the meta model is appropriate and can be used to model an example system, e.g., if all necessary model concepts are available. Second, by comparing the predicted values to measured values of an implemented architecture, we can evaluate if the introduced parametric specifications can be used to give realistic predictions. The latter are presented in the next section. This section briefly gives some details on the implementation of our simulation.

5.1 Technical foundation

The simulation is implemented as a stand-alone Java application using the Java simulation framework Desmo-J [22]. The simulation reads an instance of the Palladio Component Model. The construction of simulated (active and passive) resources and loading the configured workload model starts the simulation. The software architecture is evaluated on the fly using visitors as depicted in the following.

5.2 Visitors

The workloads stored in the model instance are transformed into simulated workload drivers, which simulate the specified workload scenario. This is done by traversing the specified usage scenario from the start action to any stop action for the specified number of clients. The behaviour of a component's service is simulated analogously traversing it from its start to its stop actions. Sensors in the simulation environment record the execution times during the simulation. The results of the simulation run are reported when the simulation ends.

Depending on the type of workload, i.e., closed workload or open workload, workload drivers are instantiated according to the respective workload's semantics. For closed workloads, the behaviour of each simulated user is executed, then the workload driver waits for the given think time and afterwards starts from the beginning. For open workloads, the workload driver starts a simulation of the workload behaviour to meet the specified arrival rate.

The traversal of the behaviour is implemented using the Visitor design pattern [10, p. 331]. A visitor visits the actions of a behaviour until it reaches a stop action. At each node simulation code is executed which simulates the specified action in this node. At control flow nodes (branch, loop, ...) the corresponding inner behaviour is executed using a new visitor started at the inner start node. At external calls the simulation looks up the respective required interface and the connected component. A new visitor is then started simulating the external service's behaviour.

At internal actions the resource demand is determined and a respective demand is put on the simulated resources. For active resource - at present - we only support a FIFO strategy. Thus, if the resource is busy, the request is put into a wait queue until the resource becomes available.

Several specifications in the SEFF like number of loop iterations, branch conditions, and resource demand depend on other random variables, i.e., parameter abstractions. A sample of this random variable is generated by the evaluation of the definition as outlined in section 4. For this, the specification of the random variable is parsed. The parse tree is evaluated afterwards. For any random variable in the parse tree the simulation framework's random number generators are used to get samples. Mathematical operations are evaluated using their normal semantics. The result of the evaluation is the desired sample of the depending random variable.

5.3 Simulated Stack-Frame

Special treatment is required for parameter abstractions, since they are only valid during the execution of the called method. To simulate this behaviour, we introduced simulated stack frames in alignment with the real execution of a software system.

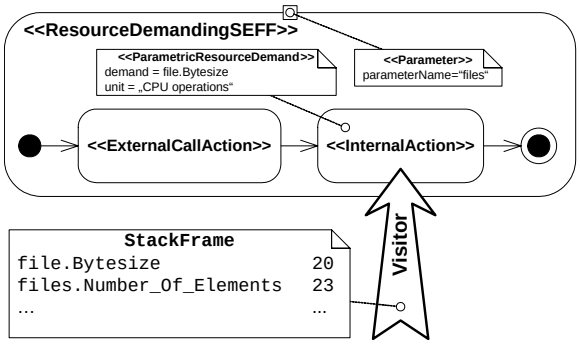

Figure 14: A behaviour visitor and the simulated stack frame

Whenever an external call is to be simulated, a new simulated stack frame is built analogously to the methods stack frame. To initialize a stack frame the random variables characterising the parameters of the called service are evaluated and the result is stored in the stack frame. It is then passed over to the visitor for the called behaviour (c.f. figure 5.3). In so doing, we simulate the performance relevant part of the usage context of the components.

When the simulated control flow returns to the caller the evaluation is continued after the external call using again the old call stack.

5.4 Simulation End

The simulation should come to an end as soon as the PMF of the workload scenario's execution time has been approximated in a way that new simulated measurements don't change the distribution significantly. This can be done by evaluating the mean and standard deviation of the resulting PMF. If the change in these characterising attributes of the PMF is below a certain configurable threshold after new simulated measurements are added, the simulation stops. Meanwhile the simulation supports live updates of its sensors, so that the system architect can watch the simulation

proceed and cancel it in advance if the results are sufficient for the analysis.

6. LIMITATIONS / ASSUMPTIONS

There are several assumptions and limitations in the current version of the Palladio Component Model and the simulation tool. We briefly summarize the most important ones in the following.

Static architecture: The modelled architecture is assumed to be static. This means that neither the connectors change nor that the components can move like agents to different hardware resources.

Abstraction from state: It is assumed that the behaviour of the system is determined by the parameters of the service calls only. No internal state of components or the runtime environment is regarded.

Information availability: Is is assumed that the necessary model information like service effect specifications and parametric dependencies are available and have been specified by the component developer, system architect, system deployer and domain expert. Future work is directed to retrieve as much information as possible from the automated analysis of component code.

Limited support for concurrency: Quality properties of concurrent systems are hard to predict. Especially on multi-core processor systems several effects like the CPU caches lead to differences between an observed system timing behaviour and an appropriate prediction in our experience.

Limited support for modelling the runtime environment: Our resource model assumes that processing resources can be described by a processing rate only. But often more than a single influence factor is important. For example, to characterize modern CPUs by the clock frequency alone, is often not sufficient any more. The CPU architecture, pipelining strategy, or the cache sizes as well as the runtime and middleware platform and their configurations can have a significant influence on the execution time (of an operation) [17].

Parameter immutability: Parameters of methods are treated as being read only during a single method execution. Hence, a parameter abstraction is not changeable during the evaluation of a single behaviour.

No return values: Service call return values are not yet modelled. Hence, for behaviours which depend on the result of an external call, the accuracy of the prediction might be insufficient.

Mathematical assumptions: Some random variables are assumed to be independent. For example, during the evaluation of a loop iterating through a collection, the characteristics of collection's inner elements are not kept constant. This limitation is subject to future model enhancements.

7. CASE STUDY

In the following, we report on an initial case study to validate predictions made by simulating instances of the Palladio component model. This case study is meant as a proof-of-concept evaluation, not as a sound experimental evaluation of the approach on an industrial sized application, which is planned for the future. We compare predictions based on architectural specifications with measurements made with an implementation of the architecture. The case study involves an online shop, which allows users uploading and downloading music files [15]. Several parameter dependencies, which will be explained in the following, can be found in the architecture, so we can test the modelling capabilities of our component model.

As we want to support early design decisions with our approach, we modelled and implemented two design alternatives for the online shop. Before the case study, we raised the following questions:

1. Are the predictions based on our simulation model good enough to support the decision for the design alternative with the actual best performance given a specified usage model?
2. Can the errors made by the predictions be quantified and explained? To answer this question, we analysed deviations between predictions and measurement in more detail.

7.1 Architecture

The *architecture* of the "Web Audio Store" (Figure 15) consists of three tiers (client, application server, database) [15]. Customers interact with the store via web browsers that access the component `WebForm` using DSL lines with a throughput of 128 KBits/s. Several components are located on the application server in the middle tier: The component `WebForm` is connected to the `AudioStore` component, which controls and manages the whole store. It interacts with a user management component and a database adapter that handles the connection to a MySQL server on the database tier. The network between the application server and the database is a dedicated line with a maximum throughput of 512 KBit/s.

As a performance critical *use case*, the response times for uploading music files to the store shall be analysed and improved. It is possible for users to upload multiple files at once to add complete music albums to the store. This usage scenario is described in Figure 16(a). In this case, users upload 8-12 files with the probabilities found in the Figure 16(b) (as "iterations=...") and files sizes between 3500 and 4500 KBytes. Upon clicking the button "Upload Files" the service `UploadFiles` of the `WebForm` component is invoked, whose behaviour is shown in Figure 16(b). This behaviour in turn invokes services from the `AudioStore` component. The parametric dependencies for the loop and the byte size of subsequent input parameters are shown in the figure. For example, the number of loop iterations depends on the number of input files. Other annotations needed for the simulation are omitted in the illustration for brevity.

Because response times of this use case are considered too high, the system architect has come up with a *design alternative*, which is shown within the dashed box in Figure 15 and which is transparent for the clients. It is proposed to insert an encoder component (`OggEncoder`) into the architecture using an adapter (`EncodingAdapter`), which implements the `IAudioDB` interface. The SEFFs of these components are illustrated in Figure 17(a)-17(b). The encoder is able to reduce the size of the music files by a factor of 2/3. Thus, the time for using the network connection to the database server can be reduced, because smaller files have to be sent. However, encoding the files is computational intensive and hence, costs an amount of time. With the performance prediction, the tradeoff between faster network transfer and reencoding overhead shall be evaluated.

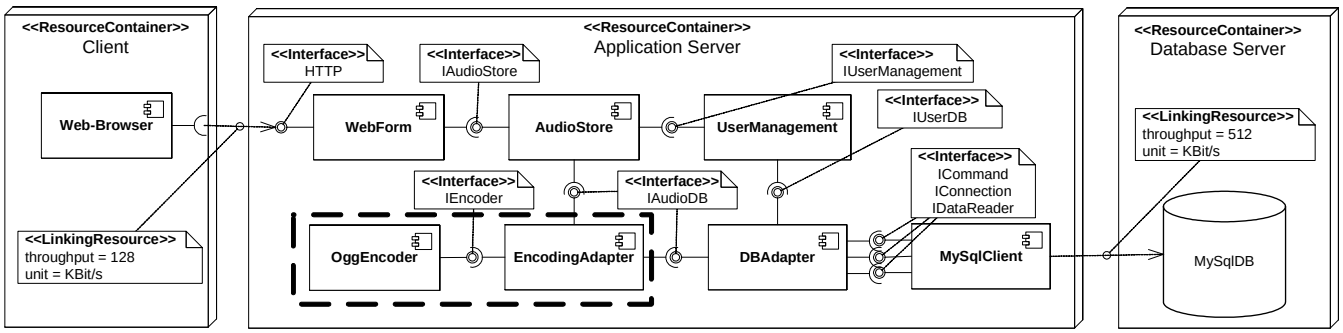

Figure 15: Web Audio Store Architecture

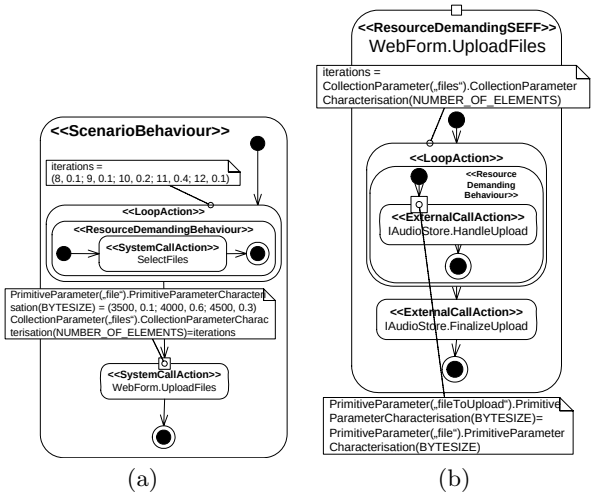

Figure 16: Original Design: Usage Model, SEFF

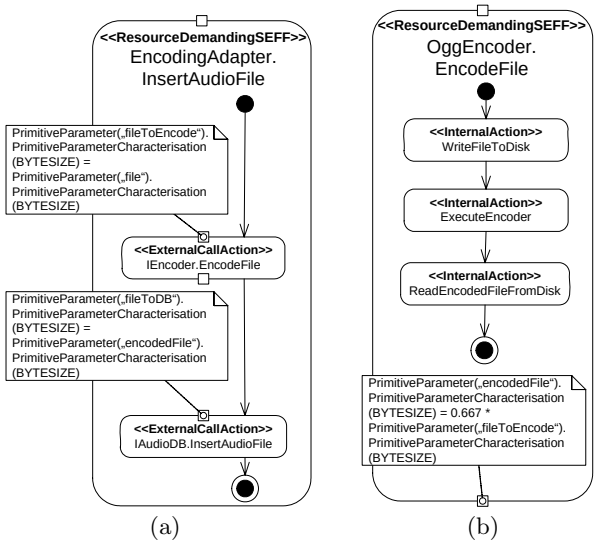

Figure 17: Design Alternative: Encoding Adapter

7.2 Results

First, we modelled and simulated both design alternatives, then we also implemented both alternatives in C# using ASP.NET. Using the workloads from the models, the response times for the described use case of the implementations were measured.

Compared to [15], we only used measured time consumptions in our prediction model for calls to the database, i.e., for submitting uncompressed files. The time consumption of submitting compressed files as well as the encoding of files had been specified parametric as introduced in the previous section. In so doing, we further weakened the assumptions which we made when we did the predictions in [15].

For the original design without encoder, the simulation results and the measured time consumptions are presented in Figure 18. To allow a visual comparison of the results, we show both histograms in a single diagram by putting them on top of each other. The simulation results are drawn in light grey and the measured results in dark grey. Overlaps of both functions result in medium grey bars.

Both functions match to a large extent. Hence, our simulation is capable to predict the response time behaviour of this design alternative based on a system model. In this variant of the architecture, the main influence on the response time is created by the amount of files in a batch upload. As we modelled the distribution of the amount of files according to what we actually used when we measured our system, this result was expected.

For the design alternative with the encoder, we first compared the simulation results of the call to `EncodeFile` with the measured values as we used a parametric dependency on the bytesize of the file to encode. The dependency was derived by a rough guess looking at a few sample encoder runs. The result is depicted in Figure 19.

It can be seen that our estimated dependency is not exactly matching, but still quite good. Using the estimated times of `EncodeFile` and the compression rate of 2/3 we simulated the whole system. The results are shown in figure 20.

Although we used the mentioned approximations of the parametric dependencies, the measured and the simulation results still match quite good. Moreover, we are able to see that the design alternative with the encoder is approximately 200 seconds faster. Our result favoured the design alternative with the encoder, which is indeed the faster one as validated by the measurements.

To answer the question whether observable prediction errors can be explained, we take a closer look at figure 20 which has such prediction errors. It can be seen that the measured

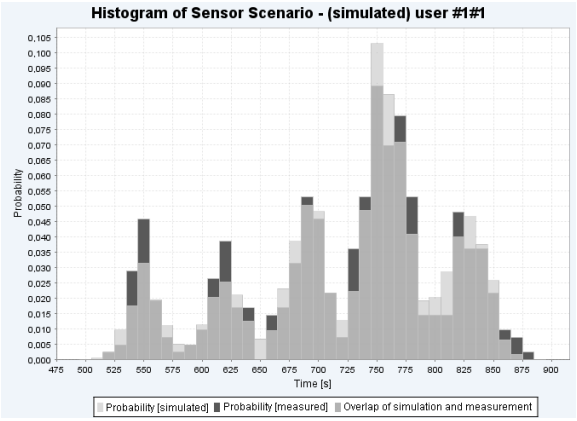

Figure 18: Measured and simulated results for the response time of UploadFiles in the design alternative without encoder

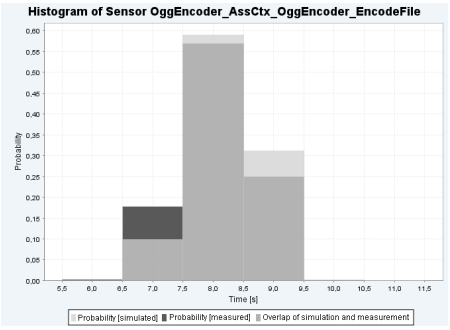

Figure 19: Measured and simulated response times of EncodeFile

values are a bit lower (approx. 10 seconds) than the predicted values. This can be explained by two issues. First, the guessed dependency for `EncodeFile` is not as accurate as it could be. A linear regression based on some measured time consumptions for different bytesizes could help to build a better estimation of the actual dependency. Second, the mathematical assumptions on the independence of random variables (c.f. section 6) is violated. The bytesize of the file being uploaded to the database depends on the bytesize of the file being sent to the encoder (we know that it is 2/3 of that size). The latter is a shortcoming of our current model abstraction and will be addressed in future work by allowing to specify correlating random variables.

8. CONCLUSIONS

This paper presents a meta-model designed to support the prediction of extra-functional properties of component based software architectures. Different possible usage contexts of a component are supported by this model. Additionally, parametric context dependencies to system resources and dependencies to parameter usages can be modelled. The soundness of the model concepts is validated by a simulation tool capable of simulating model instances. The results of the simulation are validated in a case study.

The presented method is designed to support early design time quality evaluations of component based software architectures. Based on models, a system architect can evaluate

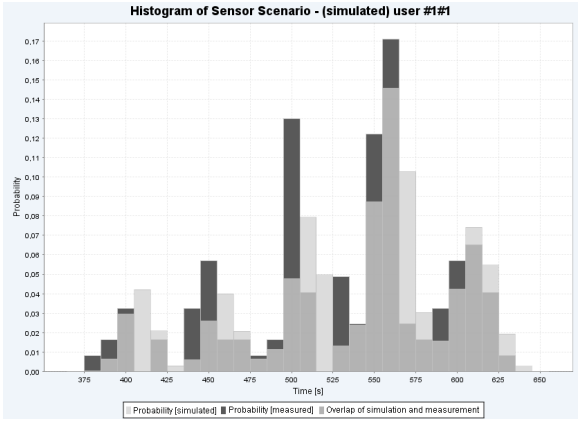

Figure 20: Measured and simulated results for the response time of UploadFiles (architecture with encoder)

the quality of the modelled system. The evaluation of design alternatives can be performed by changing the input models and re-running the simulation tool. The focus on system models enables a quick feedback cycle by the use of MDD ideas.

Our work will be extended in the future into several directions. The Palladio Component Model serves as a basis for further model transformations. Especially the generation of code skeletons for different industrial component models is an important research topic. Additionally, we try to generate application prototypes from model instances which offer similar QoS properties by enriching the aforementioned code skeletons with time consuming dummy functions.

The simulation tool is implemented as a prototype. This prototype can be extended to support all model concepts of our component model and to address some of the known limitations. Especially for systems with a lot of concurrency the simulation can serve as a basis for experiments and comparisons with running systems.

The specification of the model concepts as well as their intended semantics is still an ongoing activity. Especially the concept of using random variables for the specification of parametric dependencies is still a fairly new concept and needs further validation. Additionally, we are developing tools to ease the creation of model instances of the Palladio Component Model by the implementation or generation of graphical editors based on the Eclipse platform. These editors allow the specification of model instances in a graphical way.

Last but not least, we also plan to extend our mathematical analysis model to support additional concepts of the Palladio Component Model. Our model which is based on stochastical regular expressions should also be included into a MDD process which allows to derive it directly from instances of the Palladio Component Model.

9. REFERENCES

[1] L. B. Arief and N. A. Speirs. A UML Tool for an Automatic Generation of Simulation Programs. In *Proceedings of the Second International Workshop on Software and Performance*, pages 71–76. ACM Press, 2000.

[2] S. Balsamo, A. D. Marco, 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.

[3] S. Balsamo and M. Marzolla. A Simulation-Based Approach to Software Performance Modeling. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 363–366. ACM Press, 2003.

[4] S. Becker, L. Grunske, R. Mirandola, and S. Overhage. Performance Prediction of Component-Based Systems: A Survey from an Engineering Perspective. In R. Reussner, J. Stafford, and C. Szyperski, editors, *Architecting Systems with Trustworthy Components*, volume 3938 of *LNCS*, pages 169–192. Springer, 2006.

[5] 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, *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] 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.

[7] V. Cortellessa. How far are we from the definition of a common software performance ontology? In *WOSP '05: Proceedings of the 5th International Workshop on Software and Performance*, pages 195–204, New York, NY, USA, 2005. ACM Press.

[8] M. de Miguel, T. Lambolais, M. Hannouz, S. Betge-Brezetz, and S. Piekarec. Uml extensions for the specification and evaluation of latency constraints in architectural models. In *WOSP '00: Proceedings of the 2nd international workshop on Software and performance*, pages 83–88, New York, NY, USA, 2000. ACM Press.

[9] E. Eskenazi, A. Fioukov, and D. Hammer. Performance prediction for component compositions. In *Proceedings of the 7th International Symposium on Component-based Software Engineering (CBSE7)*, 2004.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995.

[11] V. Grassi, R. Mirandola, and A. 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] D. Hamlet, D. Mason, and D. 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] H. Koziolek and V. Firus. Parametric Performance Contracts: Non-Markovian Loop Modelling and an Experimental Evaluation. In *Proceedings of FESCA2006*, Electronical Notes in Computer Science (ENTCS), 2006.

[14] H. Koziolek and J. Happe. A Quality of Service Driven Development Process Model for Component-based Software Systems. In *Component-Based Software Engineering*, volume 4063 of *Lecture Notes in Computer Science*. Springer-Verlag GmbH, July 2006.

[15] H. Koziolek, J. Happe, and S. Becker. Parameter dependent performance specification of software components. In *Proceedings of the Second International Conference on Quality of Software Architectures (QoSA2006)*, number To appear in LNCS. Springer-Verlag, Berlin, Germany, 2006.

[16] K.-K. Lau and Z. Wang. A Taxonomy of Software Component Models. In *Proceedings of the 31st EUROMICRO Conference*, pages 88–95. IEEE Computer Society Press, 2005.

[17] Y. Liu, A. Fekete, and I. Gorton. Design-Level Performance Prediction of Component-Based Applications. *IEEE Transactions on Software Engineering*, 31(11):928–941, 2005.

[18] Object Management Group (OMG). UML Profile for Schedulability, Performance and Time. `http://www.omg.org/cgi-bin/doc?formal/2005-01-02`, January 2005.

[19] R. H. Reussner, I. H. Poernomo, and H. W. Schmidt. Reasoning on Software Architectures with Contractually Specified Components. In A. Cechich, M. Piattini, and A. Vallecillo, editors, *Component-Based Software Quality: Methods and Techniques*, number 2693 in LNCS, pages 287–325. Springer-Verlag, Berlin, Germany, 2003.

[20] M. Sitaraman, G. Kuczycki, J. Krone, W. F. Ogden, and A. Reddy. Performance Specification of Software Components. In *Proceedings of the 2001 symposium on Software reusability: putting software reuse in context*, 2001.

[21] C. U. Smith and L. G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, 2002.

[22] University of Hamburg, Department of Computer Science. The DESMO-J Homepage. `http://asi-www.informatik.uni-hamburg.de/desmoj/`.

[23] M. Woodside, D. C. Petriu, H. Shen, T. Israr, and J. Merseguer. Performance by unified model analysis (PUMA). In *WOSP '05: Proceedings of the 5th International Workshop on Software and Performance*, pages 1–12, New York, NY, USA, 2005. ACM Press.