Evaluating Performance of Software Architecture Models with the Palladio Component Model

Heiko Koziolek**, **Jens Happe
Uhlhornsweg, D-26111 Oldenburg, Germany,
heiko.koziolek@informatik.uni-oldenburg.de
jens.happe@informatik.uni-oldenburg.de

Steffen Becker, Ralf Reussner
Am Fasanengarten 5a, D-76137 Karlsruhe, Germany,
sbecker@ipd.uka.de
reussner@ipd.uka.de

Abstract
Techniques from model-driven software development are useful to analyse the performance of a software architecture during early development stages. Design models of software models can be transformed into analytical or simulation models, which enable analyzing the response times, throughput, and resource utilization of a system before starting the implementation. This chapter provides an overview of the Palladio Component Model (PCM), a special modeling language targeted at model-driven performance predictions. The PCM is accompanied by several model transformations, which derive stochastic regular expressions, queuing network models, or Java source code from a software design model. Software architects can use the results of the analytical models to evaluate the feasibility of performance requirements, identify performance bottlenecks, and support architectural design decisions quantitatively. The chapter provides a case study with a component-based software architecture to illustrate the performance prediction process.

1. Introduction (1 Page)

To ensure the quality of a software model, developers need not only to check its functional properties, but also assure that extra-functional requirements of the system can be fulfilled in an implementation of the model. Extra-functional properties include performance, reliability, availability, security, safety, maintainability, portability, etc. Like functional correctness, these properties need to be addressed already during early development stages at the model level to avoid possible later costs for redesign and reimplementation.

Performance (i.e., response time, throughput, and resource utilization) is an extra-functional property critical for many business information systems. Web-based information systems rely on fast response times and must be capable of serve thousands of users in a short time span due to the competitive nature of internet businesses. Furthermore, the responsiveness of software used within companies is important to ensure efficient business processes.

Performance problems in large distributed systems can sometimes not be solved by adding more servers with improved efficiently hardware ("kill it with iron"). Large software architectures often do not scale linearly with the available resources, but instead include performance bottlenecks that limit the impact of additional hardware.

Therefore, it is necessary to design a software architecture carefully and analyse performance issues as early as possible. However, in the software industry, performance investigations of software systems are often deferred until an implementation of the system has been build and measurements can be conducted ("fix it later"). To avoid this approach, which might lead to expensive redesigns, software architects can use performance models for early, pre-implementation performance analysis of their architectures.

This chapter provides an overview of the Palladio Component Model (PCM), a domain specific modelling language for component-based software architectures, which is specifically tuned to enable early life-cycle performance predictions. Different developer roles can use the PCM to model the software design and its targeted resource environment. The models can be fed into performance analysis tools to derive response time, throughput, and resource utilization for different usage scenarios. Software architects can use this information to revise their architectures and quantitatively support their design decisions at the architectural level.

The chapter is structured as follows: Section 2 provides background and describes related work in the area of model-driven performance prediction. Section 3 introduces different developer roles and a process model for model-driven performance predictions. Section 4 gives an overview of the PCM with several artificial model examples, before Section 5 briefly surveys different model transformations to analysis models and source code. Section 6 describes

the performance prediction for an example component-based software architecture and discusses the value of the results for a software architect. For researchers interested working in the area of model-driven performance prediction, Section 7 highlights some directions for future research. Section 8 concludes the chapter.

2. Background & Related Work (2 Pages)

Model-driven performance predictions aim at improving the quality of software architectures during early development stages (Smith et al., (2002)). Software architects use models of such prediction approaches to evaluate the response time, throughput, or resource utilization to be expected after implementing their envisioned design. The prediction model's evaluation results enable analysing different architectural designs and validate extra-functional requirements (such as a maximum response time or a minimum throughput) of software systems. The advantage of using prediction models instead of implementation testing is the lowered risk to find performance problems in already implemented systems, which require cost-intensive redesigns.

Researchers have put much effort into creating accurate performance prediction models for the last 30 years. Queuing networks, stochastic process algebras, and stochastic Petri nets are the most prominent prediction models from the research community. However, practitioners seldom apply these models due to their complexity and high learning curve. Therefore, focus of the research community has shifted to create more developer-friendly models and use model transformations to bridge the semantic gap to the above mentioned analytical models.

From the more than 20 approaches in this direction during the last decade (Balsamo et al., (2004)), most use annotated UML models as a design model and ad-hoc transformations to create (layered) queuing networks as analytical models. Tools shall encapsulate the transformation to the analytical models and their solution algorithms to limit the necessary additional skills for designers. For these approaches, the Object Management Group (OMG) has published multiple UML profiles (SPT, QoS/FT, MARTE) to add performance-related annotations to UML models. However, these profiles remain under revision, are still immature, and are still not known to have been used in practise in a broader scope.

Component-based software engineering (CBSE) adds a new dimension to model-driven performance prediction approaches. CBSE originally targeted at improved reusability, more flexibility, cost-saving, and shorter time-to-market of software systems. Besides these advantages, CBSE might also ease prediction of extra-functional properties. Software developers may test components for reuse more thoroughly and provide them with more detailed

specifications. These specifications may contain performance-related information.

Hence, several research approaches have tackled the challenge of specifying the performance of a software component (cf. survey by Becker et al., (2006)). This is a difficult task, as the performance of a component depends on environmental factors, which can and should not be known by component developers in advance. These factors include:

- *Execution Environment*: the platform a component is deployed on including component container, application server, virtual machine, operating system, software resources, hardware resources
- *Usage Profile*: user inputs to component services and the overall number of user requests directed at the components
- *Required Services*: execution times of additionally required, external services, which add up to the execution of the component itself

Component developer can only fix the component's implementation, but have to provide a performance specification, which is parameterisable for the execution environment, the usage profile, and the performance of required services. The following paragraph summarises some of the approaches into this direction.

Sitaraman et. al (2001) model the performance of components with an extension to the O-calculus, but do not include calls to required services. Hissam et. al (2002) aim at providing methods to certify component for their performance properties. Bertolino et. al (2003) use the UML-SPT profile to model component-based systems. They explicitly include dependencies to the execution environment, but neglect influences by the usage profile. Hamlet et al. (2003) investigate the influence of the usage profile on component performance. Wu et al. (2004) model components with an XML-based language and transform this notation into layered queueing networks. The APPEAR method by Eskenazi et al. (2004) aims at predicting performance for changes on already built systems, and thus does neglect the influence of the execution environment. Bondarev et al. (2005) target components in embedded systems with their ROBOCOP model. Grassi et al. (2005) develop an intermediate modelling language for component-based systems called KLAPER, which shall bridge the gap between different design and analytical models.

The Palladio Component Model (Becker et al., (2007)) described in this chapter is in line with these research approaches and tries to reflect all influences on component performance. Unlike some of the above listed approaches, the PCM does not use annotated UML as design model, but defines its own metamodel. This reduces the model to concepts necessary for performance prediction and does not introduce the high complexity of arbitrary UML models with a variety of concepts and views.

3. Developer Roles and Process Model (2 Pages)

The PCM metamodel is divided into several domain-specific modelling languages, which are aligned with developer roles in CBSE. This section introduces these roles and provides an overview of the process model for using the PCM.

An advantage of CBSE is the division of work between different developer roles, such as component developers and software architects. *Component developers* specify and implement components. They also have to provide a description of the component's extra-functional properties to enable software architects to predict their performance without deploying and testing them. *Software architects* compose components from different component developers to application architectures. They are supported by tools to predict the architecture's performance based on the performance specifications of the component developers. With the predicted performance metrics, they can support their design decisions for different architectural styles or components.

For performance predictions, the software architect needs additional information about the execution environment and the usage profile. The role of the *system deployer* provides performance-related information about the hardware/software environment of the architecture (such as speed of a CPU, throughput of a network link, scheduling policies of the operating system, configuration of the application server, etc.). Business *domain experts* mainly possess knowledge about the anticipated user behavior (in terms of input parameters and call frequencies), and must assists software architects in specifying an usage model of the architecture.

Figure 1 depicts the overall development process of a component-based system including performance prediction (Koziolek et al. (2006)).

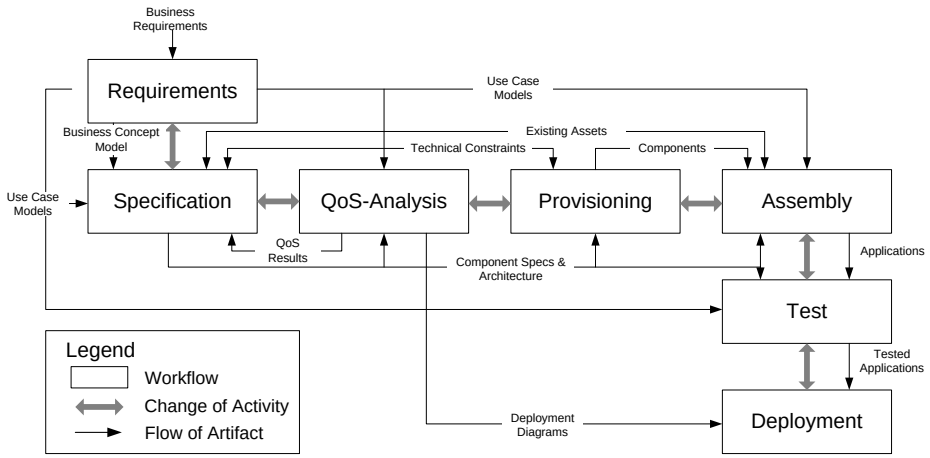

Figure 1: Component-based Development Process

Boxes model workflows, thick, grey arrows indicate a change of activity, and thin, black arrows illustrate the flow of artifacts. The workflows do not have to be traversed linearly; backward steps for revision are likely. After collecting and analysing requirements for the system to develop (Requirements), the software architect specifies components and the architecture based on input by component developers (Specification). With a fully specified architecture, performance predictions can be carried out by tools (QoS-analysis). The software architect can use the results to alter the specification or decide to implement the architecture. This is done either by obtaining existing components from third-party vendors or by implementing them according to their specification (Provisioning). Afterwards, the software architect can compose the component implementations (Assembly), test the full application in a restricted environment (Test), and then install and operate it in the customer's actual environment (Deployment).

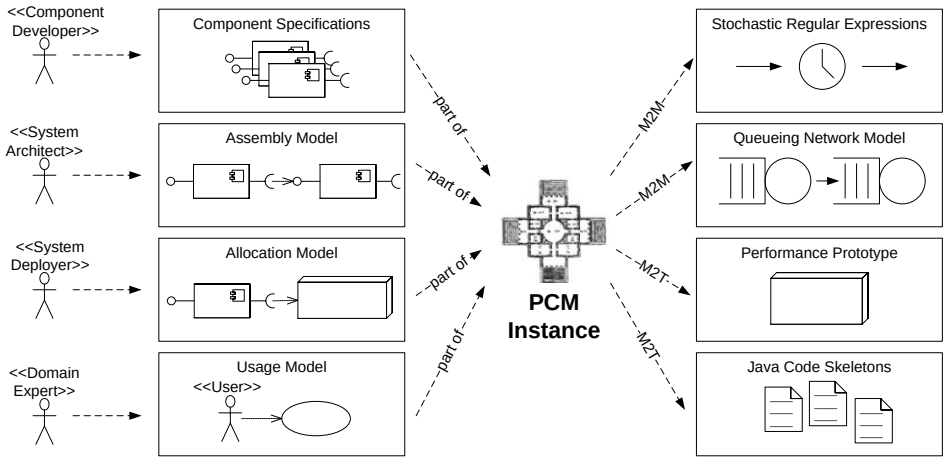

Figure 2: Specification and QoS Analysis with the PCM

During "Specification", the above introduced roles interact as follows (cf. Figure 2): The PCM provides a domain-specific modelling language for each developer role, which is restricted to concepts known to this role. Component developers model performance-related component behaviour, software architects add an assembly model. System deployers model hardware/software resources and the components' allocation to these resources. Finally, domain experts provide a usage model. All specifications can be combined to derive a full PCM instance. Section 4 will elaborate on the PCM's specification languages.

During "QoS-Analysis", this model can be transformed into different analysis models, such as stochastic regular expressions or a queueing network. These

models provide capabilities to derive performance metrics such as response times, throughputs, or resource utilisations for specific usage scenarios. Additionally, the PCM can be transformed into a performance prototype, which simulates the specified resource demands. This prototype enables pre-implementation performance measurements on the target platform. Finally, the PCM instance can be converted into Java code skeletons via Model2Text transformation, as a starting point for implementing the system's business logic. Section 5 describes the analysis models and code transformations in more detail.

4. Overview Palladio Component Model (10 Pages)

This section provides an overview of the modeling capabilities of the PCM to describe component-based software architecture. The PCM is a metamodel specified in EMF/Ecore. The following section will mainly use examples to introduce the concepts, and does not go into technical details of the metamodel, which the reader can find in (Reussner et al., 2007). The description of the PCM in this section is structured along the developer roles and their domain-specific languages.

4.1 Component Developer (6 Pages)

Component developers specify the functional and extra-functional properties of their components. They put the specification as well as the implementation in repositories, where software architects can retrieve them. This section will first introduce all entities, which can be stored in repositories and then focus on service effect specifications, which model the abstract behavior and performance properties of component services.

4.1.1 Component Repositories

Figure 3 shows an example PCM repository, which includes all types of entities that can be specified. First class entities in PCM repositories are interfaces, data types, and components. They may exist on their own and do not depend on other entities.

The *interface* MyInterface is depicted on the upper left in Figure 3. It is not yet bound to a component, but can be associated as a provided or required interface to components. An example of interfaces existing without clients and an implementation in practice was the Java Security API, which had been specified by Sun before an implementation was available. Interfaces in the PCM contain a list of service signatures, whose syntax is based on CORBA IDL. Additionally, component developers may supplement an interface with protocols, which restrict the order of calling its services. For example, an I/O interface might force clients to first open a file (call service open()) before reading from it (call service read()).

Components may provide or require interfaces. The binding between a component and an interface is called "provided role" or "required role" in the PCM. For example, component A in Figure 3 is bound to YourInterface in a provided role. This means that the component includes an implementation for each of the services declared in the interface. Other components, which are bound to a compliant interface in a required role can use component A to execute these services.

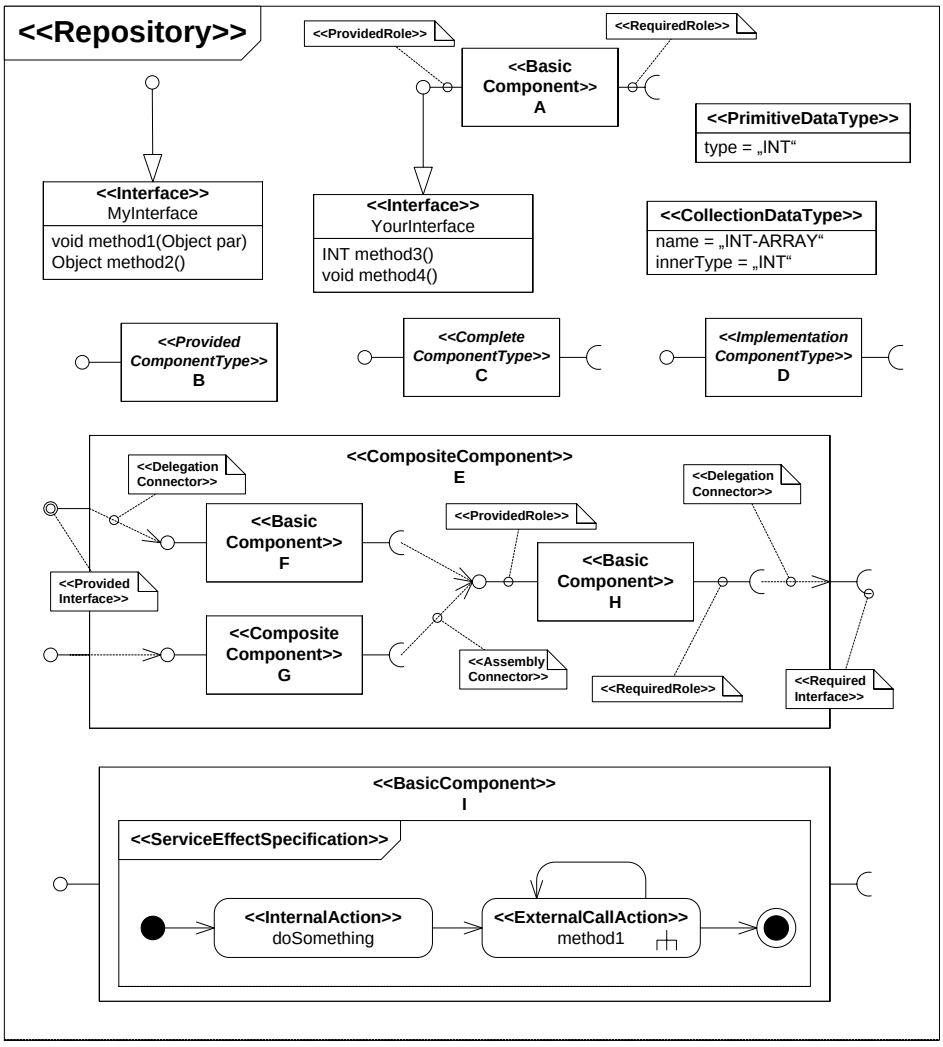

Figure 3: Example Component Repository

Repositories need common data types, so that the service signatures refer to standardized types (e.g., INT, FLOAT, CHAR, BOOL, STRING, etc.). In the PCM, data types are either primitive types, collection types, or composite types (composed out of inner types). Figure 3 contains a primitive data type INT and a collection data type INT-Array, which contains INTs as inner elements.

The PCM supports modeling different types of components to a) reflect different development stages, and b) to differentiate between basic (atomic) components and composite components.

Different development stages are reflected by *provided, complete, and implementation component type*. Component developers can refine components during design from provided to implementation component types.

Provided component types (component B in Figure 3) only provide one or more interfaces, but include no mandatory required interfaces. Component developers can use these type of components early during the development, when they know that a certain functionality has to be provided, but do not know whether other components are needed to provide this functionality.

Complete component types (component C in Figure 3) are provided component types, but additionally may contain mandatory required interfaces. However, the inner dependencies between provided and required interfaces are not fixed in complete component types, as different implementations can lead to different dependencies. Within a component architecture, a software architect may easily replace one component with another component, which conforms to the same complete component type, without affecting the system's functionality.

Implementation component types (component D in Figure 3) are complete component types, but additionally contain fixed inner dependencies between provided and required interfaces. Replacing implementation component types in an architecture ensures not only signature but also protocol compatibility at the required interface.

Implementation component types are either basic or composite components. Component E in Figure 3 is a *composite component*. It contains several inner components (F, G, H). Inner component may again be composite components (G) to build up arbitrary hierarchies. Assembly connectors bind the roles of inner components. Delegation connectors connect provided roles of composite components with provided roles of inner components, or required roles of composite components with required roles of inner components. From the outside, composite components look like basic components, as they provide and require services. The inner structure of a composite component should only be known to the component developer, but not to the software architect, who shall use the component as a unit and treat it the same as other components.

Finally, basic components are "standard" components, which cannot be further decomposed. They may contain a mapping for each provided service to required services, which is called resource demanding service effect specification.

4.1.2 Service Effect Specification

Resource demanding service effect specifications (RDSEFF) provide means to describe resource demands and calls to required services by a provided component service. Component developers use RDSEFFs to specify the performance of their components.

RDSEFFs reflect the environmental factors on component performance introduced in Section 2. They are parameterisable for external services, execution environment, usage profile, and component implementation as described in the following subsection.

RDSEFFs abstractly model the externally observable behavior of a component service. They only refer to method signatures and parameters that are declared in the interfaces and add control flow between calls to required services, parametric dependencies, and resource usage. These specifications do not reveal any additional knowledge about the algorithms used to implement the service's functionality and thus retain the black-box principle.

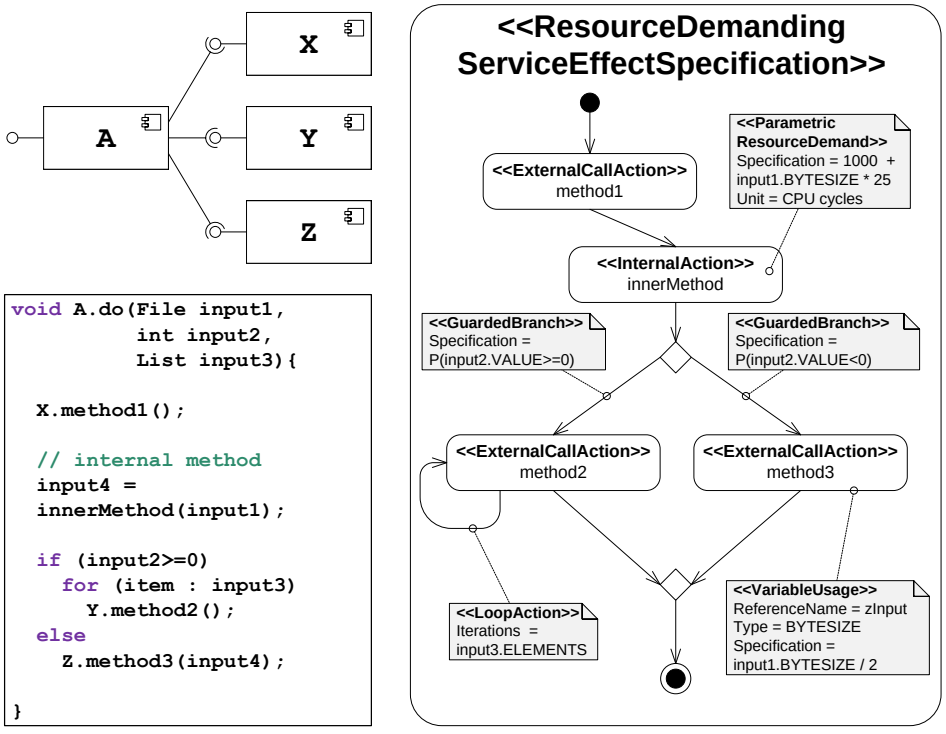

Figure 4: Resource Demanding Service Effect Specification

Consider the artifical example in Figure 4 for a brief introduction into RDSEFFs. Component A invokes required functionality via its required X,Y,

and Z. It provides a service called "do", whose source code is sketched in Figure 4. The service first calls a service from interface X, and then executes some internal code processing parameter "input1". Afterwards, depending on "input2", either services from interface Y or Z are called. "method2" from interface Y is located within a loop, whose number of iteration depends on the array length of "input3".

The corresponding RDSEFF for service "do" is located on the right hand side in Figure 4. As a graphical, concrete syntax, the illustration uses the UML activity diagram notation. However, in this case, the metamodel underlying the modeling constructs is not the UML metamodel, but the PCM, which is indicated by stereotypes (within << >>). In the following, the underlying concepts for control flow, resource demands, and parametric dependencies will be described.

Control Flow: Actions in RDSEFFs can either be internal actions (i.e., the component executes some internal code) or external call actions (i.e., the component calls a service declared in its required interface). The order of these actions may influence performance properties of the service, because different services may concurrently use the same resources or synchronize each other, which induces delays for waiting. Hence, RDSEFF offer as basic control flow constructs sequences, alternatives, loops, and parallel executions (forks).

Alternatives or branches split the control flow with an XOR semantic, while forks (not depicted in Figure 4) split the control flow with an AND semantic, i.e., all following actions are executed concurrently. Loops have to specify the number of iterations, so that the execution times for actions within the loop can be added up a limited number of times.

Notice that the control flow in RDSEFFs is an abstraction from the actual inner control flow of the service. Internal actions potentially summarize a large number of inner computations and control flow constructs, which do not contain calls to required services.

Resource Demands: Besides external services, a component service accesses the resources of the execution environment it is deployed in. Ideally, component developers would provide measured execution times for these resource accesses in the RDSEFF. However, these measured times would be useless for software architects, who want to use the component, because their hardware/software environment can be vastly different from the component developer ones. The execution times of the service could be much faster or slower in the software architect's environment.

Therefore, component developers specify resource demands in RDSEFFs against abstract resource types such as a CPU or hard disk. For example they can provide the number of CPU cycles needed for execution or the number of bytes read from or written to a hard disk. The resource environment model supplied by the system deployer (Section 4.3) then contains execution times for

executing CPU cycles or reading a byte from hard disk. These values can be used to calculate the actual execution times of the resource demands supplied by the component developers. As an example, the "ParametricResourceDemand" on the internal action "method1" in Figure 4 specifies that the service needs 1000 CPU cycles plus the amount of a parametric dependency (described below) to execute.

In addition to active resources, such as processors, storage devices, and network devices, component services may also acquire and release passive resources, such as threads, semaphores, database connections etc. Passive resources are not capable of processing requests and usually exist only a limited number of times. A service can only continue its execution, if at least one of them is available. Acquisition and release of passive resources is not depicted in Figure 4.

Parametric Dependencies: To include the influence of the usage profile into the RDSEFF, component developers can specify parametric dependencies. When specifying an RDSEFF, component developers cannot know how the component will be used by third parties. Thus they cannot fix resource demands, branching probabilities or the number of loop iterations if those values depend on input parameters. Hence, RDSEFFs allow specifying dependencies to input parameters.

There are several forms of these dependencies. For example, in Figure 4, the resource demand of the internal action "innerMethod" depends on byte size of input parameter "input1" (e.g., because the method processes the file byte-wise). Once the domain expert characterizes the actual size of this parameter (cf. Section 4.4), this value can be used to calculate the internal action's actual resource demand.

Furthermore, branching probabilities are needed for the alternative execution paths in this RDSEFF. These probabilities are however not fixed, but depend on the value of input parameter "input2". Therefore, the RDSEFF includes no branching probabilities but guards (i.e., Boolean expressions) on the branches. Once the domain expert characterizes the possible values of "input2" and provides probabilities for the input domains "input2<=0" and "input2>0", these values can be mapped to the branching probabilities.

The RDSEFF in Figure 4 also contains a parametric dependency on the number of loop iterations surrounding the external call to "method2" of component Y. Loop iterations can be fixed in the code, but sometimes they depend on input parameters. In this case the service iterates over the list "input3" and calls the external service for each of its elements. The RDSEFF specifies this dependency as the component developer cannot know in advance the lengths of the lists.

Finally, the service "do" executes the external call to "method3" in Figure 4 with an input parameter that in turn depends on an input parameter of the service itself. The service processes "input1", assigns it to a local variable

"input4", and then forwards it to interface Z via "method3". While processing "input1", the service "do" reduces its byte size by 50% ("input1.BYTESIZE / 2"). The RDSEFF includes the specification of this dependency. Once the domain expert specifies the actual byte size of "input1", the byte size of the input parameter of "method3" can be calculated.

4.2 Software Architect (1 Page)

Software architects retrieve components (including their RDSEFFs) from repositories and compose them to architectures. They can use several component instances of the same type in an architecture at different places. Hence, in the PCM, software architects put component instances in so called assembly contexts, which are representations of a single component instance and its connected provided and required roles.

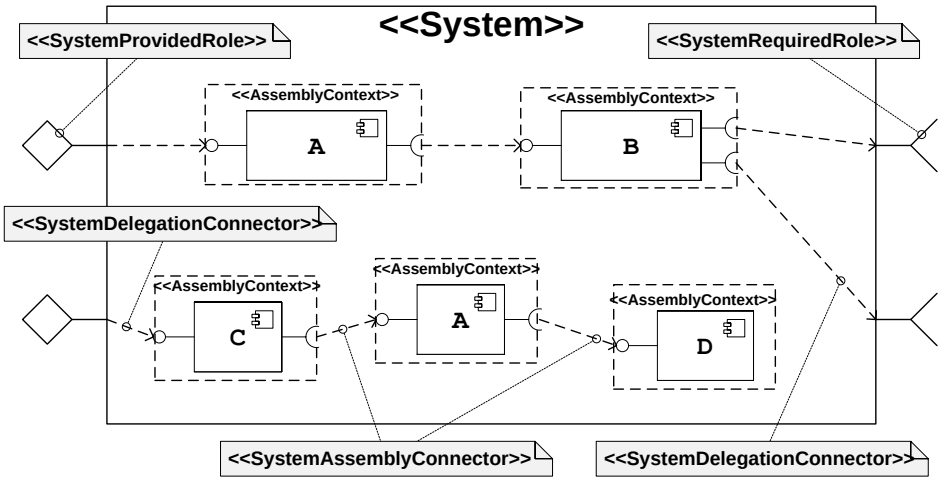

Figure 5: System Example

Software architects bind the roles of components in assembly contexts with system assembly connectors, as illustrated in the example in Figure 5. Notice that the component type A is used in two assembly contexts in this example (once connected with component B and once connected with components C and D).

A set of connected assembly contexts is called *assembly*. An assembly is part of a *system*, which additionally exposes system provided roles and system required roles (cf. Figure 5). System delegation connectors bind these system roles with roles of the system's inner components. Domain experts later use system provided roles to model the usage of the system (Section 4.4). System required roles model external services, which the software architect does not consider

part of the architecture. For example, the software architect can decide to model a web service or a connected database as system external services.

There is a distinction between composite components and systems. For software architects and system deployers, but not for component developers, composite components hide their inner structure and the fact that they are composed from other components. The inner structure is an implementation detail and its exposure would violate the information hiding principle of components. Opposed to this, the structure of assemblies is visible to software architects and system deployers. Therefore, system deployers can allocate each component in a system to a different resource. However, they cannot allocate inner components of composite components to different resources, because these stay hidden from them at the architectural level.

4.3 System Deployer (1 Page)

System deployers first specify the system's resource environment and then allocate assembly contexts (i.e., connected component instances) to resources.

In *resource environments*, resource containers group resources. For example, in Figure 6, the resource container "Server1" contains a CPU, a hard disk, and a database connection pool. The PCM differentiates between processing resources, which can execute requests (e.g., CPU, hard disk, memory), and passive resources, which cannot execute requests, but only be acquired and released (e.g., threads, semaphores, database connections).

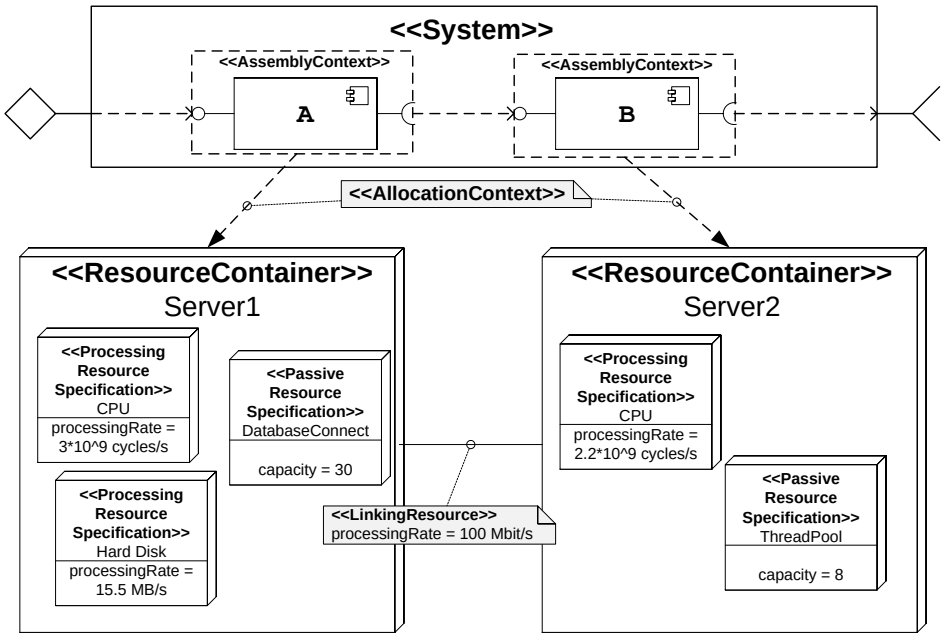

Figure 6: Resource Environment and Allocation

Processing resources specify a processing rate, which can be used to convert the resource demands in RDSEFFs into timing values. Passive resources specify a capacity. If a component acquires a passive resource, its amount of available units (i.e., its capacity) decreases. Once the capacity reaches zero, further components requesting the passive resource must wait until other services release it again. Linking resources connect resource containers and are themselves special processing resources.

System deployers use *allocation contexts* to specify that a resource container executes an assembly context. In Figure 6, the system deployer has allocated component A's assembly context to "Sever1" and component B's assembly context to "Server2".

System deployers can specify different resource environments and different allocation contexts to answer sizing questions. The PCM's resource model is still limited to abstract hardware resources. We will extend it in the future with middleware parameter, operating system settings, and scheduling policies.

4.4 Domain Expert (1 Page)

Domain experts create a usage model that characterizes user behavior and connects to system provided roles. In the example in Figure 7, users first log in to the system, then either browse or search, then buy an item, and finally log out. All actions target system provided roles (i.e., services exposed by the system, cf. Section 4.2).

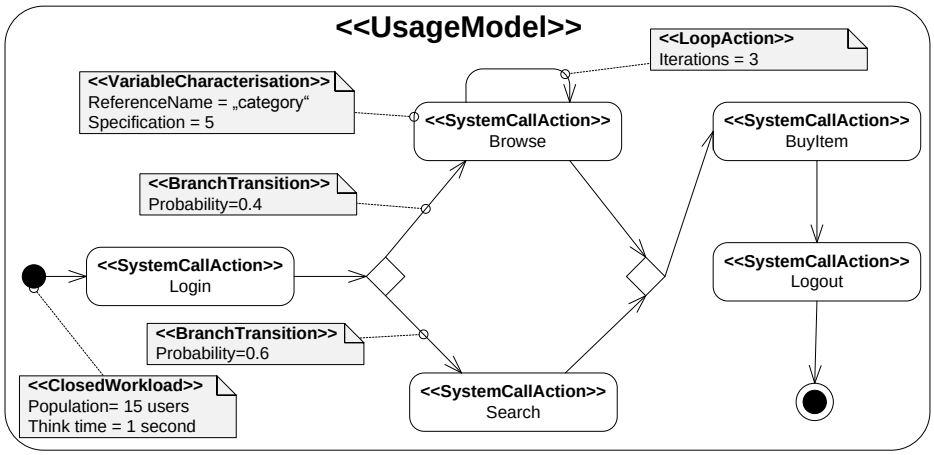

Figure 7: Usage Model Example

Domain experts can specify user behavior with control flow constructs such as sequence, alternative, and loop. They must specify branching probabilities for alternatives and the number of iterations for loops.

Additionally, domain experts specify the user workload. Workloads are either closed or open. Closed workloads specify a fixed number of users (population) circulating in the system. In Figure 7, the domain expert has specified a closed workload with 15 users, which perform the specified actions and then re-enter the system after a think time of 1 second. Open workloads specify a user arrival rate (e.g., 5 users/second), and do not limit the number of users in the system.

The PCM usage model also enables domain experts to characterize the parameter values of users. In Figure 7, variable "category" of action browse has been characterized with a constant (5) meaning that users always browse in the category with id number 5. Besides constants, the usage model offers specifying probability distribution functions over the input domain of a parameter, so that domain experts can provide a fine-grained stochastic characterization of the user's input parameters. The reader may find details in Reussner et al. (2007).

4.5 Tool Support (1 Page)

We have implemented an Eclipse-based open-source tool called "PCM-Bench", which enables software developers to create instances of the PCM metamodel and run performance analyses (cf. Figure 8). The tool offers a different view perspective for each of the four developer roles and provides graphical model editors. The PCM-Bench is an Eclipse RCP application and its editors have been partially generated from the PCM Ecore metamodel with help of the Graphical Modelling Framework (GMF).

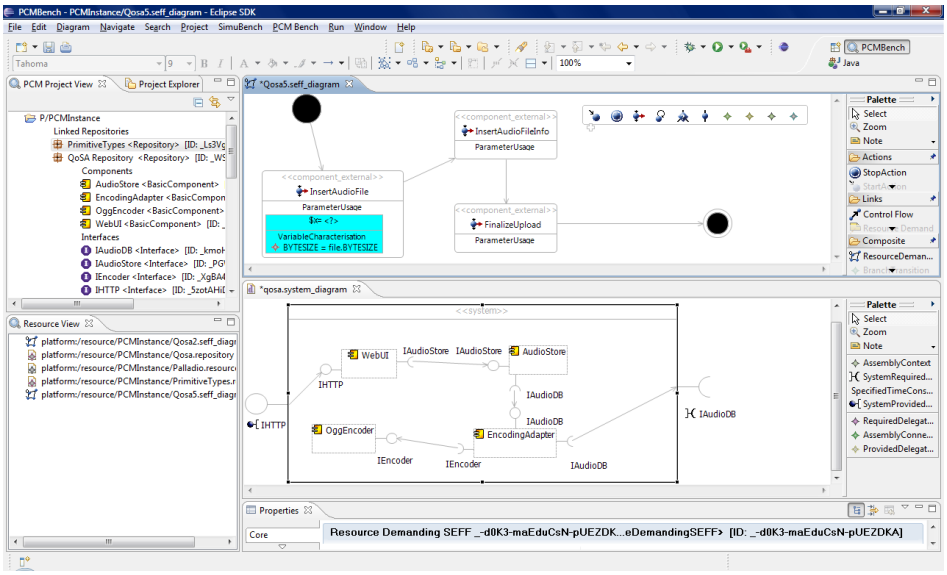

Figure 8: Screenshot PCM-Bench

The graphical editors provide an intuitive way of modeling component-based architectures analogous to UML modeling tools. They offer model validation by checking OCL-constraints. The PCM-Bench visualizes violated constraints directly in the model diagrams. The editors support entering performance annotations with special input masks that offer syntax highlighting and code completion. Model instances can be serialized to XMI-files.

Besides graphical editors, the PCM-Bench is a front-end for the performance analysis techniques described in Section 5. Software architects can configure and run simulations. They can retrieve different performance metrics such as response times for use cases, throughputs, and resource utilizations. The PCM-Bench visualizes probability distribution functions of response times as histograms and provides standard statistical values such as mean, median, standard deviation etc. Furthermore, the PCM-Bench supports Model-to-Text transformations to generate Java code from PCM instances.

5. Model Transformation and Prediction Methods (5 Pages)

The PCM offers different performance evaluation techniques, which are still subject to research. For analyzing use cases without concurrency, a PCM instance can be transformed into a stochastic regular expression (SRE), which offers a fast way of predicting response times in presence of resource demands specified as general distribution functions (Section 5.1). For cases with multiple users, a PCM instance can be transformed into a queuing network based simulation model (Section 5.2). The simulation model is less restricted than the SREs, but its execution is usually more time consuming than solving the SREs. Finally, there are transformations to derive Java code skeletons from a PCM instance, to provide a starting point for implementing the modeled architecture (Section 5.3).

5.1 Stochastic Regular Expressions (1 Page)

To transform a PCM instance into a SRE, tools first solve the parametric dependencies within the SEFFs. The tools use the parameter characterizations provided by the domain expert in the usage model to transform parametric resource demands to resource demands, guards on branches to probabilities, and parameter dependent loop iteration specifications into iteration numbers. Afterwards, the transformation into SREs is straightforward (Koziolek et. al (2007a)). The following briefly describes the syntax, semantics, and calculations with SREs, which derive response times for use cases as probability density functions.

The syntax of SREs is specified as follows (BNF): $P := a \mid P \cdot Q \mid P +_{\pi} Q \mid P^{*(l)}$, where “a” denotes a symbol containing a random variable X_a for an execution time, “$P \cdot Q$” denotes a sequence, “$P +_{\pi} Q$” denotes an alternative with probabilities π and $1-\pi$, and $P^{*l(n)}$ denotes a loop.

An independent and identically distributed probability density function (PDF) $f_a(t)$ characterizes the execution time of random variable X_a.

The execution time of a sequence of two expressions "P · Q" is the sum of the random variables $X_{P \cdot Q}=X_P + X_Q$, whose distribution is determined by the convolution of their PDFs: $f_{P \cdot Q}(t) = f_p(t)$ (*) $f_Q(t)$.

For the alternative "$P +_\pi Q$" with probabilities π and $1-\pi$, the resulting PDF is the weighted (by branching probabilities) sum of the expressions' PDFs: $f_{P +\pi Q}(t) = \pi f_p(t) + 1-\pi f_Q(t)$.

The execution time of a loop $P^{*l(n)}$ depends on the number of loop iterations, which is specified by a probability mass function (PMF) $p_l(i) = P_l(X = i)$ denoting the probability that expression P is executed i times. The PDF of a loop is computed by a weighted convolution: $f_{P^{*l(n)}}(t) = \Sigma\, p_l(i)\, (\, {}^{*}\, f_p)\, (t)$.

These basic constructs are sufficient to derive the PDF of a response time of a complete use case if no concurrent behavior is executed. We use Fast Fourier Transformation to efficiently calculate the convolution of density functions. The SRE metamodel is, like the PCM metamodel, specified in Ecore. So far, we have implemented the model-to-model transformation from a PCM instance to a SRE instance in Java. For the future, we plan a QVT-based transformation. The reader may find details of SREs and their underlying assumptions in Koziolek et al. (2007).

5.2 Queuing Network Simulation (1 Page)

Many performance analysis methods use queuing networks as underlying prediction models because of their capability to analyze concurrent system interactions. Queuing models contain a network of service centers with waiting queues which process jobs moving through the network. When applying queuing networks in performance predictions with the PCM, some of the commonly used assumptions need to be dropped. As the PCM uses arbitrary distribution functions for the random variables, generalized distributed service center service times, arrival rates, etc. occur in the model. Additionally, the requests travel through the queuing network according to the control flow specified in the RDSEFF. In contrast, common queuing networks assume probabilistic movement of the jobs in the network. As a result, only simulation approaches exist, which solve such models.

Hence, we use a model-to-text transformation to generate Java code realizing a custom queuing network simulation based on the simulation framework Desmo-J. The simulation generates service centers and their queues for each active resource. Passive resources are mapped on semaphores initialized with the resource's capacity. The transformation generates Java classes for the components and their assembly. Service implementations reflect their respective SEFF.

For the usage model workload drivers for open or closed workloads simulating the behavior of users exist in the generated code. For any call issued to the simulated system, the simulation determines the parameter characterizations of the input parameters and passes them in a so called virtual stackframe to the called service. Originally, the concept of a stackframe comes from compiler construction where they are used to pass parameters to method calls. In the PCM simulation, stackframes pass the parameter characterizations instead.

Utilizing the information in the simulated stackframes, the simulated SEFF issues resource demands to the simulated resources. If the resource is contented, the waiting time increases the processing time of the demand.

The simulation runs until simulation time reaches a predefined upper limit or until the width of the estimation for the confidence interval of the mean of any of the measured response times is smaller than a predefined width. After the end of a simulation run, the simulation result contains different performance indicators (response times, queue lengths, throughputs …) which the software architect can analyze to determine performance bottlenecks in the software architecture.

5.3 Java Code & Performance Prototype (1 Page)

The effort spent into creating a model of a software architecture should be preserved when implementing the system. For this, a model-to-text transformation based on the openArchitectureWare (oAW) framework generates code skeletons from PCM model instances. The implementation uses either Plain Old Java Objects (POJOs) or Enterprise Java Beans (EJBs) ready for deployment on a J2EE application server.

The transformation uses as much model information as possible for the generation of artifacts. Repository models result in components, system assemblies in distributed method calls, the allocation is used to generate ant scripts to distribute the components to their host environment and finally, the usage model results in test drivers.

A particular challenge is the mapping of concepts available in the PCM to objects used in Java or EJB. Consider for example the mapping of composite components to Java. As there is no direct support of composed structures in Java, a common solution to encapsulate functionality is the application of the session façade design pattern.

Another issue with classes as implementing entities for components is the missing capabilities to explicitly specify required interfaces of classes in object oriented languages. A solution for this is the application of the component context pattern by Völter et al. (2006). This pattern moves the references to required services into a context object. This object is injected into the component either by an explicit method call or by a dependency injection mechanism offered by the application server.

Finally, we can combine the EJB and the simulation transformation. This way, users can generate a prototype implementation which can be readily deployed and tested on the final execution environment. Internal actions of the prototype only simulate resource demands by executing dummy code which offers quality characteristics as specified in the model. By using the prototype, early simulation results can be validated on the real target environment to validate early performance estimates.

6. Example (3 Pages)

To illustrate the performance prediction approach with the PCM, this section provides a case study, in which we predicted the response time of a usage scenario in a component-based software architecture and compared the results with measured response times from executing an implementation.

The system under analysis is the "MediaStore" architecture, a web-based store for purchasing audio and video files, whose functionality is modeled after Apple's iTunes music store. It is a three-tier architecture assembled from a number of independently usable software components (Figure 9). Users interact with the store via web browsers, and may purchase and download different kinds of media files, which are stored in a database connected to the store's application server via Gigabit Ethernet.

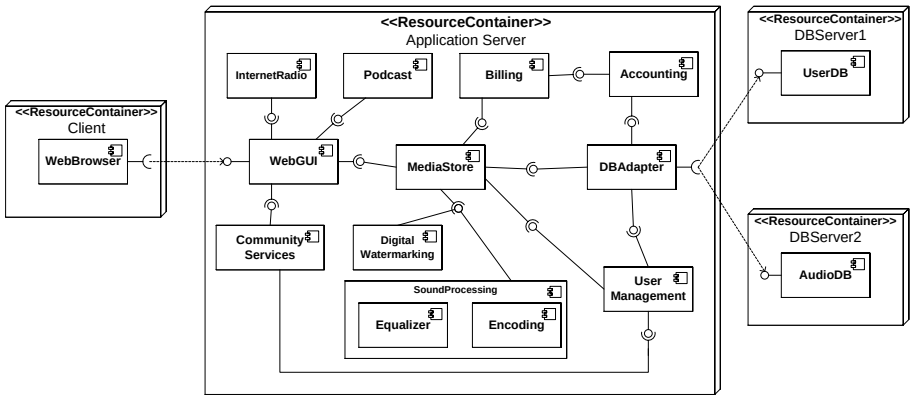

Figure 9: MediaStore Architecture

We analysed a scenario, in which users purchase a music album (10-14 files, 2-12 MB per file) from the store. As a measure for copy protection, a component "DigitalWatermarking" shall be incorporated into the store. This component unrecognisable attaches the user's ID to the audio files via digital watermarking. In case the audio files would appear illegally in public file sharing services, this enables tracking down the responsible user. However, this copy protection measure has an influence on performance, as it decreases the response time of the store when downloading files. With the model-driven performance

prediction, we analysed whether the store is capable of answering 90% of all user download requests in less than 8 seconds.

Each component in the store provides RDSEFFs to enable performance analyses (three examples in Figure 10). The execution time in this use case mainly depends on the number and size of the files selected for download, which influences network traffic as well as CPU utilisation for the watermarking algorithm. The specifications of the the components' RDSEFFs have been calibrated with measurements on the individual components. In this case, we carried out the predictions using SREs.

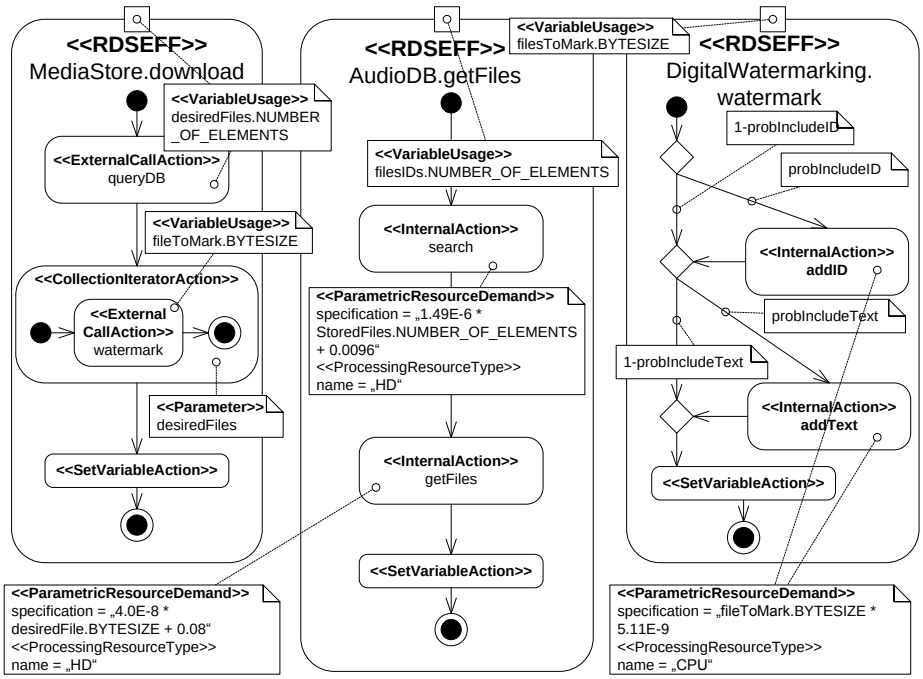

Figure 10: MediaStore Service Effect Specifications

Besides modelling the store, we also implemented the architecture assisted by the introduced model-to-text transformations to Java code (EJB3). After generating code skeletons from the design, we manually added the implementation of the business logic of forwarding requests and watermarking audio files. The code generation also creates build scripts, test drivers, deployment descriptors, and configuration files. We weaved measurement probes into the code using AspectJ.

The results of prediction and measurement are compared in Figure 11. The diagram on the left hand side visualises the histograms of the response times. The dark columns indicate the prediction, while the bright columns on top of

the dark columns indicate the measurement. The highest probability of receiving a response from the store with the mentioned parameters is at around 6 second. In this case, the prediction and the measurement widely overlap.

The diagram on the right hand side visualises the cumulative distribution functions of the response time prediction and measurements. This illustration allows to easily check our constraint of at least 90% of all responses in less than 8 seconds. It was predicted that 90% of all requests would be responded in 7.8 seconds even if watermarking was used in the architecture. The measurements confirmed the predictions, because in our tests 90% of the request could be answered less than 7.8 seconds. There is a difference of 0.1 seconds or 1.3 percent.

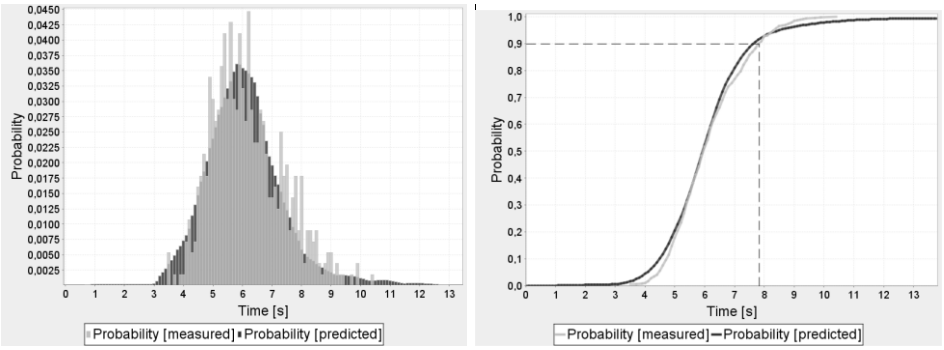

Figure 11: Case Study Results

In this case, the predictions were useful to quantitatively support the software architect's decision to introduce watermarking without violating a service level agreement. Note, that the predictions are not meant to be real-time predictions for safety-critical systems. They are useful at early development stages on the architectural level to support design decisions and lower the risk of performance problems in implementations. Safety-critical systems (e.g., airbag controls) instead need formal verifications of predictions to prevent harming human lives. That requires more fine grain specifications at lower abstraction levels, which developers can only create if most details of the system are known.

7. Future Research Directions (350-500 words)

Model-driven performance prediction and quality assurance of software architecture models is still in its infancy and provides lots of opportunities for future research. Woodside et al. (2007) recently commented on the future of software performance engineering. We provide a list of future research directions from our viewpoint complementing their ideas:

- **Intermediate Languages:** To bridge the gap between designer-friendly model notations and analytically-oriented formalisms, many approaches have developed ad-hoc model transformations. Several approaches aim at providing a standard interface, i.e., an intermediate modelling language, to ease the implementation of model transformations (Grassi et al. (2005), Petriu et. al. (2005))
- **Dynamic Architectures**: The PCM is only targeted at static architectures, and does not allow the creation/deletion of components during runtime or changing links between components. With the advent of web services and systems with dynamic architectures changing during runtime, researchers pursuit methods to predict the dynamic performance properties of such systems (Caporuscio et al. (2007), Grassi et al. (2007)).
- **Layered Resource Models:** With OMG's MDA vision of platform independent models and platform specific models, it seems straight forward to follow this approach in performance modelling. For different system layers (e.g., component containers, middleware, virtual machine, operating system, hardware resources), individual models capturing performance-relevant properties could be built. These models could be composed with architectural models to predict the performance (Woodside et. al (2007)).
- **Combination of Modeling and Measurement:** Developers can only carry out performance measurements if the system or at least parts of it have been implemented. Measurement results could be used to improve models. In component-based performance modelling, measurements are useful to deduce the resource demands of components. A convergence of early-life cycle modelling and late-life cycle measurement can potentially increase the value of performance evaluations (Woodside et. al (2007)).
- **Performance Engineering Knowledge Database:** Information collected by using prediction models or measuring prototypes tends to get lost during system development. However, the information is useful for future maintenance and evolution of systems. Systematic storage of performance-related information in a knowledge database could improve performance engineering (Woodside et. al (2007)).
- **Improved Automated Feedback:** While today's model-transformations in software performance engineering bridge the semantic gap from the developer-oriented models to the analytical models, the opposite direction of interpreting performance result back from the analytical models to the developer-oriented models has received sparse attention. Analytical performance results tend to be hard to interpret by developers, who lack knowledge about the underlying formalisms. Thus, an intuitive feedback from the analytical models to the developer-oriented models would be appreciated (OMG (2005), Woodside et. al (2007)).

8. Conclusion

This chapter provided an overview of the Palladio Component Model, a modelling language to describe component-based software architectures aiming at early life cycle performance predictions. The PCM is aligned with developer roles in CBSE, namely component developers, software architects, system deployers, and domain experts. Therefore, the PCM provides a domain specific modelling language for each of these developer roles. Combining the models from the roles leads to a full PCM instance specification, which can be transformed to different analysis models or Java code. An analytical model (SRE) provides a fast way to predict response times in single-user scenarios. Simulation of PCM instances is potentially more time-consuming, but offers support for multi-user scenarios. Finally, developers may use generated Java code skeletons from a PCM instance as a starting point for implementation. To illustrate the PCM's capabilities the chapter included a case study predicting the performance for a small component-based architecture.

The PCM is useful both for component developers and software architects. Component developers can specify the performance of their components in a context-independent way, thereby enabling third party performance predictions and improving reusability. Software architects can retrieve component performance specification from repositories and assemble them to architectures. With the specifications they can quickly analyse the expected performance of their designs without writing code. This lowers the risk of performance problems in implemented architectures, which are a result of a poor architectural design. The approach potentially saves large amounts of money because of avoided re-designs and re-implementations.

The chapter provided pointers for future directions of the discipline in Section 7. Future work for the PCM includes improving the resource model, supporting dynamic architectures and reverse engineering. The resource model needs to be improved to support different scheduling disciplines, concurrency patterns, middleware parameters, operating system features etc. Dynamic architectures complicate the model as they allow changing links between components and allow the creation and deletion of components during runtime. However, this is common in modern service-based systems, and thus should be incorporated into performance predictions. Finally, reverse engineering to semi-automatically deduce performance models from existing legacy code seems an interesting pointer for future research. Reducing the effort for modelling would convince more developers of applying performance predictions. The inclusion of legacy systems enables predicting the impact on performance of planned system changes.

References

Balsamo, S. , DiMarco, A., Inverardi, P. & Simeoni, M. (2004). Model-Based Performance Prediction in Software Development: A Survey. *IEEE Transactions on Software Engineering, 30*(5), 295-310.

Becker, S.; Grunske, L.; Mirandola, R. & Overhage, S. (2005). Performance Prediction of Component-Based Systems: A Survey from an Engineering Perspective. In *Springer Lecture Notes in Computer Science Vol. 3938* (pp. 169-192).

Becker, S., Koziolek, H. & Reussner, R. (2007). Model-based Performance Prediction with the Palladio Component Model. In *Proceedings of the 6th Workshop on Software and Performance WOSP'07* (pp. 56-67). ACM Press

Bertolino, A. & Mirandola, R. (2004). CB-SPE Tool: Putting Component-Based Performance Engineering into Practice. In Crnkovic, I., Stafford, J. A., Schmidt, H. W. & Wallnau, K. C. (Ed.), *Proceedings of the 7th International Symposium on Component-Based Software Engineering, CBSE2004* (pp. 233-248). Springer Lecture Notes in Computer Science, Vol. 3054

Bondarev, E., de With, P., Chaudron, M. & Musken, J. (2005). Modelling of Input-Parameter Dependency for Performance Predictions of Component-Based Embedded Systems. In *Proceedings of the 31th EUROMICRO Conference (EUROMICRO'05)*

Caporuscio, M., DiMarco, A. & Inverardi, P. (2007), Model-based system reconfiguration for dynamic performance management. *Journal of Systems and Software, 80*(4), (pp. 455-473). Elsevier

Eskenazi, E., Fioukov, A. & Hammer, D. (2004). Performance Prediction for Component Compositions. In Crnkovic, I., Stafford, J. A., Schmidt, H. W. & Wallnau, K. C. (Ed.), *Proceedings of the 7th International Symposium on Component-Based Software Engineering, CBSE2004*. Springer Lecture Notes in Computer Science, Vol. 3054

Grassi, V., Mirandola, R. & Sabetta, A. (2005). From design to analysis models: a kernel language for performance and reliability analysis of component-based systems. In *Proceedings of the 5th international workshop on Software and performance, WOSP '05* (pp. 25-36). ACM Press

Grassi, V., Mirandola, R. & Sabetta, A. (2007). A Model-Driven Approach to Performability Analysis of Dynamically Reconfigurable Component-Based

Systems. In *Proceedings of the 6th international workshop on Software and performance, WOSP '07* (pp. 142-153). ACM Press

Hamlet, D., Mason, D. & Woit, D. (2004). Properties of Software Systems Synthesized from Components. In Lau, K. (Ed.), *Component-Based Software Development: Case Studies* (pp. 129-159). World Scientific Publishing Company

Hissam, S. A., Moreno, G. A., Stafford, J. A. & Wallnau, K. C. (2002). Packaging Predictable Assembly. In *CD'02: Proceedings of the IFIP/ACM Working Conference on Component Deployment* (pp. 108-124). Springer-Verlag

Koziolek, H., Happe, J. & Becker, S. (2006). Parameter Dependent Performance Specifications of Software Components. In Hofmeister, C., Crnkovic, I., Reussner, R. & Becker, S. (Ed.) *Proceedings of the 2nd International Conference on the Quality of Software Architecture, QoSA2006* (pp. 163-179). Springer Lecture Notes in Computer Science, Vol. 4214

Koziolek, H., Happe, J. & Becker, S. (2007). Predicting the Performance of Component-based Software Architectures with different Usage Profiles. In Szyperski, C. & Overhage, S. (Ed.) *Proceedings of the 3rd International Conference on the Quality of Software Architecture, QoSA2007*. Springer Lecture Notes in Computer Science

Petriu, D. B. & Woodside, M. (2005). An intermediate metamodel with scenarios and resources for generating performance models from UML designs. *Springer Journal on Software and Systems Modeling*

Reussner. R. H., Becker, S., Happe, J., Koziolek, H., Krogmann, K. & Kuperberg. M. (2007). *The Palladio Component Model*. Internal Report Universität Karlsruhe (TH)

Sitaraman, M., Kuczycki, G., Krone, J., Ogden, W.F. & Reddy, A. (2001). Performance Specifications of Software Components. In *Proceedings of the Symposium on Software Reusability 2001* (pp. 3-10).

Szyperski, C., Gruntz, D. & Murer, S. (2002). *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley

Wu, X. & Woodside, M. (2004). Performance modeling from software components. In *Proceedings of the 4th International Workshop on Software*

Performance, WOSP2004 (pp. 290-301). ACM SIGSOFT Software Engineering Notes

Völter, M. & Stahl, M. (2006). *Model-driven Software Development*. Wiley & Sons

Woodside, M., Franks, G. & Petriu D. C. (2007). The Future of Software Performance Engineering. In *Proceedings of 29th International. Conference on Software Engineering, ICSE'07.* Track: Future of Software Engineering.

Additional Reading (25-50 References)

Bolch, G., Greiner, S., de Meer, H. & Trivedi, K. S. (2006). *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications.* Wiley-Interscience, 2nd Edition

Cecchet, E., Marguerite, J. & Zwaenepoel, W.(2002) Performance and scalability of EJB applications. *ACM SIGPLAN Notes, 37*(11), 246-261

Chen, S., Liu, Y., Gorton, I. & Liu, A. (2005). Performance prediction of component-based applications. *Journal of Systems and Software, 74*(1), 35-43.

DiMarco, A. & Mirandola, R. (2006). Model transformations in Software Performance Engineering. *Springer Lecture Notes in Computer Science,* Vol. 4214, 95-110

Dumke, R., Rautenstrauch, C., Schmietendorf, A. & Scholz, A. (2001). *Performance Engineering: State of the Art and Current Trends.* Springer Lecture Notes in Computer Science, Vol. 2047

Grassi, V., Mirandola, R. & Sabetta, A. (2006). Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach. *Journal of Systems and Software, 80*(4), 528-558.

Hermanns, H., Herzog, U. & Katoen, J. (2002) Process algebra for performance evaluation. *Theorectical Computer Science, 274*(1-2), Elsevier Science Publishers Ltd., 43-87

Jain, R. K. (1991). *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley

Kounev, S. (2006). Performance Modeling and Evaluation of Distributed Component-Based Systems Using Queueing Petri Nets. *IEEE Transactions on Software Engineering, 32*(7), 486-502.

Lazowska, E.; Zahorjan, J.; Graham, G. & Sevcik, K. (1984). *Quantitative System Performance,* Prentice Hall

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

Menasce, D. A. & Gomaa, H. (2000). A Method for Design and Performance Modeling of Client/Server Systems. *IEEE Transactions on Software Engineering, 26*(11), 1066-1085

Menasce, D. A. & Almeida, V. A.(2000) *Scaling for E-Business: Technologies, Models, Performance, and Capacity Planning,* Prentice Hall

Menasce, D. A. & Almeida, V. A.(2002) *Capacity Planning for Web Services,* Prentice Hall

Menasce, D. A., Dowdy, L. W. & Almeida, A.F. (2004). *Performance by Design: Computer Capacity Planning By Example*, Prentice Hall PTR

Reussner, R. H., Schmidt, H. W. & Poernomo, I. H. (2003). Reliability prediction for component-based software architectures. *Journal of Systems and Software, 66*(3), 241-252.

Rolia, J. A. & Sevcik, K. C. (1995). The Method of Layers. *IEEE Transactions on Software Engineering, 21*(8), 689-700

Smith, C. U. & Williams, L. G. (2001). *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software.* Addison-Wesley Professional

Verdickt, T., Dhoedt, B., Gielen, F. & Demeester, P (2005). Automatic Inclusion of Middleware Performance Attributes into Architectural UML Software Models. *IEEE Transactions on Software Engineering, 31*(8), 695-771.

Woodside, C. M., Neilson, J. E., Petriu, D. C. & Majumdar, S. (1995) The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software. *IEEE Transactions on Computers,* 44(1), 20-34.