Performance Evaluation of Component-based Software Systems: A Survey

Heiko Koziolek[a]

[a]*ABB Corporate Research, Industrial Software Systems, Wallstadter Str. 59, 68526 Ladenburg, Germany*

Abstract

Performance prediction and measurement approaches for component-based software systems help software architects to evaluate their systems based on component performance specifications created by component developers. Integrating classical performance models such as queueing networks, stochastic Petri nets, or stochastic process algebras, these approaches additionally exploit benefits of component-based software engineering, such as reuse and division of work. Although researchers have proposed many approaches into this direction during the last decade, none of them has attained widespread industrial use. On this behalf, we have conducted a comprehensive state-of-the-art survey of more than 20 of these approaches assessing their applicability. We classified the approaches according to the expressiveness of their component performance modelling languages. Our survey helps practitioners to select an appropriate approach and scientists to identify interesting topics for future research.

Key words: Performance, Software Component, CBSE, Prediction, Modelling, Measurement, Survey, Classification

1. Introduction

During the last ten years, researchers have proposed many approaches for evaluating the *performance* (i.e., response time, throughput, resource utilisation) of component-based software systems. These approaches deal with both performance prediction and performance measurement. The former ones analyse the expected performance of a component-based software design to avoid performance problems in the system implementation, which can lead to substantial costs for re-designing the component-based software architecture. The latter ones analyse the observable performance of implemented and running component-based systems to understand their performance properties, to determine their maximum capacity, identify performance-critical components, and to remove performance bottlenecks.

Component-based software engineering (CBSE) is the successor of object-oriented software development [85, 41] and has been supported by commercial component frameworks such as Microsoft's COM, Sun's EJB, or CORBA CCM. *Software components* are units of composition with explicitly defined provided and required interfaces [85]. Software architects compose software components based on specifications from component developers. While classical performance models such as queueing networks [57], stochastic Petri nets [3], or stochastic process algebras [42] can be used to model and analyse component-based systems, specialised component performance models are required to support the component-based software development process and to exploit the benefits of the component paradigm, such as reuse and division of work. The challenge for component performance models is that the performance of a software component in a running system depends on the context it is deployed into and its usage profile, which is usually unknown to the component developer creating the model of an individual component.

The goal of this paper is to classify the performance *prediction and measurement approaches* for component-based software systems proposed during the last ten years. Beyond tuning guides for commercial component frameworks (e.g., [86, 100, 17]), which are currently used in practise, these approaches introduce specialised modelling languages for the performance of software components and aim at an understanding of the performance of a designed architecture instead of code-centric performance fixes. In the future, software architects shall predict the performance of application designs based on the component performance specifications by component developers. This approach is more cost-effective than fixing late life-cycle performance problems [91].

Although many approaches have been proposed, they use different component notions (e.g., EJB, mathematical functions, etc.), aim at different life-cycle stages (e.g., design, maintenance, etc.), target different technical domains (e.g., embedded systems, distributed systems, etc.), and offer different degrees of tool support (e.g., textual modelling, graphical modelling, automated performance simulation). None of the approaches has gained widespread industrial use due to the still immature component performance models, limited tool support, and missing large-scale case-study reports. Many software companies still rely on personal experience and hands-on approaches to deal with performance problems instead of using engineering methods [94]. Therefore, this paper presents a survey and critical evaluation of the proposed approaches to help selecting an appropriate approach for a given scenario.

Our survey is more detailed and up-to-date compared to existing survey papers and has a special focus on component-based performance evaluation methods. Balsamo et al. [1] re-

Email address: heiko.koziolek@de.abb.com (Heiko Koziolek)

viewed model-based performance prediction methods for general systems, but did not analyse the special requirements for component-based systems. Putrycz et al. [75] analysed different techniques for performance evaluation of COTS systems, such as layered queueing, curve-fitting, or simulation, but did not give an assessment of component performance modelling languages. Becker et al. [5] provided an overview of component-based performance modelling and measurements methods and coarsely evaluated them for different properties. Woodside et al. [94] designed a roadmap for future research in the domain of software performance engineering and recommended to exploit techniques from Model-Driven Development [84] for performance evaluation of component-based systems.

The *contributions* of this paper are (i) a classification scheme for performance evaluation methods based on the expressiveness of their component performance modelling language, (ii) a critical evaluation of existing approaches for their benefits and drawbacks, and (iii) an analysis of future research directions. We exclude qualitative architectural evaluation methods, such as ATAM [50] and SAAM [49] from our survey, because they do not provide quantitative measures. We also exclude approaches without tool support or case studies (e.g., [22, 30]). Furthermore, we exclude performance modelling and measurement approaches, where monolithic modelling techniques have been used to assess the performance of component-based systems (e.g., [40, 52]).

This paper is structured as follows. Section 2 lays the foundations and elaborates on the special characteristics of performance evaluation methods for component-based systems. Section 3 summarises the most important methods in this area. Section 4 discusses general features of the approaches and classifies and evaluates them. Section 5 points out directions for future research. Section 6 concludes the paper.

2. Software Component Performance

This section introduces the most important terms and notions to understand the challenge of performance evaluation methods for component-based systems. Section 2.1 clarifies the notion of a software component, before Section 2.2 lists the different influence factors on the performance of a software component. Section 2.3 distinguishes between several life-cycle stages of a software component and breaks down which influence factors are known at which life-cycle stages. Based on this, Section 2.4 derives the requirements for a component performance modelling language, which are expressed through a feature model. Finally, Section 2.5 shows a generic process model for performance evaluation of component-based system, which defines the scope of the surveyed methods.

2.1. Software Components

McIlroy [61] coined the term 'software component' already at the 1968 NATO conference on software engineering. However, the concept of reusable software components did not get widespread attention until the mid-nineties, when the foundations became more clear [85, 41] and commercial component frameworks, such as Microsoft's COM [16], Sun's EJB [24] and OMG's CCM [69] appeared. Lau and Wang have surveyed a large number of component system models [56]. Szyperski defines a software component as follows [85]:

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

Software components manifest the principles of information hiding and separation of concerns. They foster reuse and prepare systems for change of individual parts. Moreover, they enable a division of work between component developers and software architects, therefore lowering the complexity of the overall development task. Black-box components only reveal their provided and required interfaces to clients, whereas white-box components allow viewing and modifying the source code of the component implementation. Composite components bundle several components into larger units.

2.2. Factors influencing Component Performance

Specifying the performance of reusable software components is difficult, because the provided performance depends not only on the component implementation, but also on the context the component is deployed into. Factors influencing the performance of software components are (Fig. 1, also see [7]):

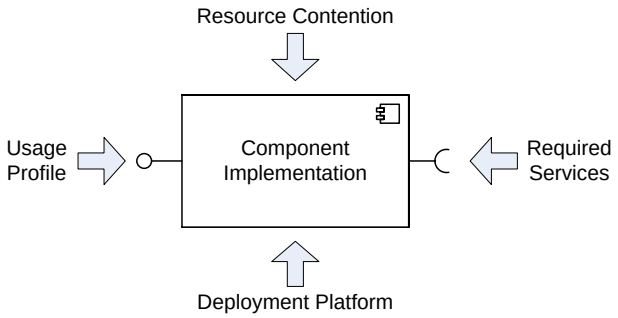

Figure 1: Factors influencing Component Performance

- **Component Implementation:** Component developers can implement the functionality specified by an interface in different ways. Two components can provide the same service functionally, but exhibit different execution times running on the same resources and given the same inputs.
- **Required Services:** When a component service A invokes required services B, the execution time of B adds up to the execution time of A. Therefore, the overall execution time of a component service depends on the execution time of required services.
- **Deployment Platform:** Different software architects deploy a software component to different platforms. A deployment platform may include several software layers (e.g., component container, virtual machine, operating

system, etc.) and hardware (e.g., processor, storage device, network, etc.).

- **Usage Profile:** Clients can invoke component services with different input parameters. The execution time of a service can change depending on the values of the input parameters. Besides input parameters of provided services, components may also receive parameters as the result of calls to required services. The values of these parameters can also influence the execution time of a service. Furthermore, components can have an internal state from initialisation or former executions, which changes execution times.

- **Resource Contention:** A software component typically does not execute as a single process in isolation on a given platform. The induced waiting times for accessing limited resources add up to the execution time of a software component.

2.3. Component Life-Cycle

The influence factors on the performance of a software component become known at different stages of its life-cycle. A performance evaluation requires all influence factors to be available. If a software architect wants to analyse a running system, these factors can be measured using tools. If a software architect wants to make a performance prediction of a component-based system at an early life-cycle stage, the influence factors fixed in later life cycle stages have to be estimated and modelled.

To clarify the notion of component life-cycle, Figure 2 shows four stages of the idealised component life-cycle by Cheesman and Daniels [14]. It refers to component specification, implementation, deployment, and runtime. We will distinguish between *functional models* for implementation and interoperability checking and *performance models* for performance requirements checking and performance prediction of a software component at each life-cycle stage.

A *specified component* is described via its provided and required interfaces. Notations for such a description are for example an interface definition language (IDL) or UML interfaces. At this stage there is still no implementation of the component. A functional model may include protocols for valid call sequences. A performance model for a specified component can only include performance requirements for the specified provided services (e.g., maximum 20 ms response time), since the implementation, deployment, and runtime information for the component is missing.

An *implemented component* realises a component specification by providing the functionality specified by the provided interfaces using only the specified required interfaces. A functional model can now include information on how the provided services of the implementation call the required interfaces (i.e., a so-called service effect specification [79]). A performance model of an implemented but not deployed and used component can now also include information about the behaviour and the resource demands of the component, however parametrised over the concrete platform and input parameters.

A *deployed component* results from assembling an implemented component to other components and allocating it onto a hardware node. At this stage, the component consists of the implementation and deployment information (e.g., an EJB or BPEL deployment descriptor). The functional model now includes information about which provided services can actually be offered, because it is now known which required services are available [79]. The performance model can now include information about the component container, operating system, and hardware. Therefore the platform-independent resource demands (e.g., processor cycles) can be transformed into platform-dependent resource demands (e.g., timing values) still parametrised for the input parameters.

Finally, a *runtime component* is instantiated and may serve client requests. Such a component is for example an object in memory. At runtime, components can have an internal state, which now can be included into the functional model to check the violation of valid protocol states. For the performance model, at this stage the workload (i.e., the number of clients calling the component), the input parameters, and the performance-relevant internal state can be added. Furthermore, the overall performance model for the system may include information about concurrently running processes also involved in resource contention.

Many approaches for performance evaluation target a performance prediction for a component-based system already at the specification stage (level 0 in Fig. 2). At this stage, the software architect has to make assumptions about the information from level 1-3. For level 1, current state-of-the art requires component developers to provide a performance model of their component implementation parametrised over deployment platform, required services, and usage profile to the software architect. For level 2, some methods use a benchmark application to measure the performance of basic services of the deployment platform. Using this information, the former parametrisation for the deployment platform can be solved. For level 3, current approaches rely on estimations of the workload, input parameters, and concurrent processes from the software architect based on former experience and customer requirements.

2.4. Requirements for a Component Performance Modelling Language

Software components and component performance models are provided by the component developer. During specification of a component, the component developer has no information about components connected to its required interfaces, its deployment platform, or parameter values passed to its provided services by clients. Because of this, the component developer has to provide a parametrised specification (i.e., a function over the different factors), which makes the influences by these external factors explicit.

For each provided service of a component, the component developer must provide a service performance specification, which should include (Fig. 3):

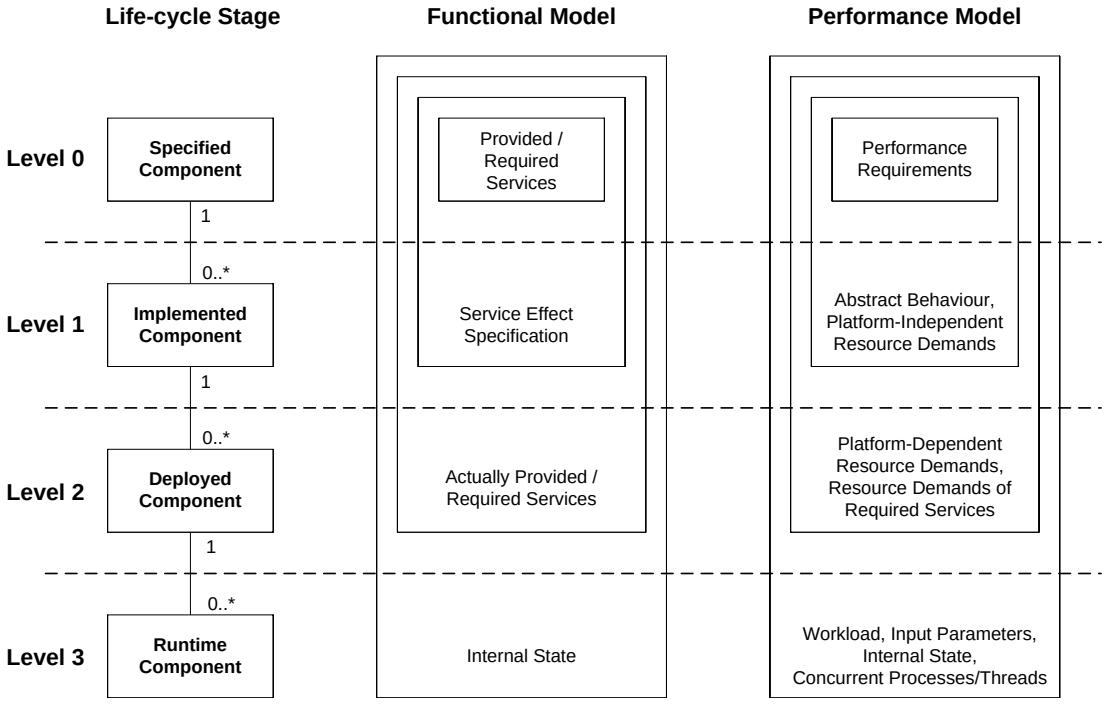

Figure 2: Software Component Life-Cycle

- **Schedulable Resource Demands:** During execution, a component service can access different active resources, such as a processor or a storage device. Each of these resources can become the bottleneck of the system due to contention effects. To find such bottlenecks, it is necessary that each component service specifies resource demands to be scheduled on each active resource. Such a demand can be specified as a constant or a distribution function. The latter is especially useful for large software components, which cannot be modelled in detail. The unit of scheduled resource demands may be either a platform-dependent timing value (e.g., seconds) or a platform-independent value (e.g., CPU cycles). The latter realises the parametrisation for different deployment platforms.
- **Limited Resource Demands:** Besides active resources, a component service might also acquire and release limited resources, such as semaphores, threads from a pool, memory buffers, etc., which might lead to waiting delays due to contention with other concurrently executed services.
- **Control Flow:** The order of accessing resources or calling required services (e.g., in sequences, alternatives, loops or forks) by a component service might change the resource contention in a component-based system and should therefore be included.
- **Required Service Calls:** The component developer must make calls to required services explicit in the service performance specification, so that their contribution to the overall execution time can be taken into account by the software architect. Required service calls can be synchronous (i.e., the caller blocks until receiving an answer) or asynchronous (i.e., the caller continues execution immediately after the call).
- **Parameter Dependencies:** The values of service parameters can change the execution time or memory consumption of a service, its accesses to active or passive resources, the number of calls to required services, and the control flow. Because the actual parameter values used by clients are unknown during component specification, component developers need to specify properties such as execution time or memory consumption in dependency to service parameter values.
- **Internal State:** A software component can have a global state (equal for all clients) or a local state (different for each client), which should be modelled abstractly if it influences the performance.

Service performance specifications potentially have to abstract from the actual performance behaviour of a component, to remain analysable in an acceptable amount of time. The component developer creating a service performance specification must make a trade-off between the accuracy of the specification and its analysability.

Service performance specifications should not refer to network devices nor threading levels. Network communication in component-based systems should only happen between components when calling required services, but not inside components. Otherwise, if a software component would require mul-

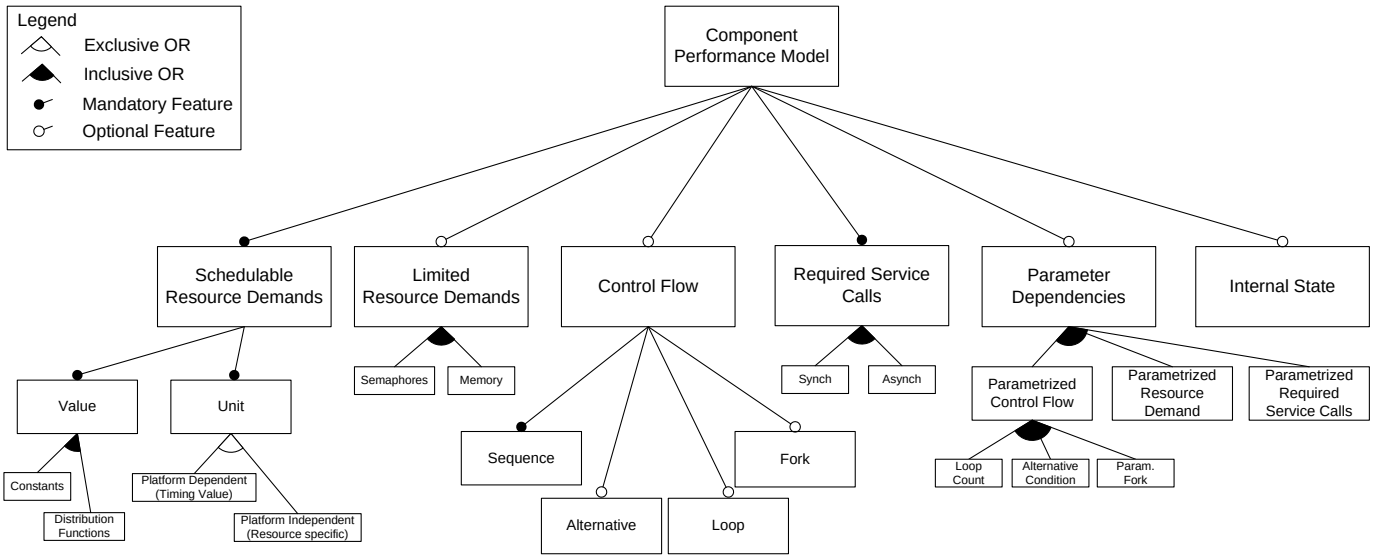

Figure 3: Feature Diagram for Component Performance Models

tiple servers connected by network devices, it would not be a unit of deployment as stated by Szyperski's definition [85]. The maximum number of threads concurrently executing a component is a major influence factor on the performance of a software component, but depends on the configuration of the middleware and not the component itself.

2.5. A Life-Cycle Process Model for Component-based Software System Performance Engineering

Performance evaluation of a component-based system involves information from the component developer and software architect. It may span from performance predictions at early design stages to performance measurements on the running application. Fig. 4 shows a generic process model for applying the methods and tools surveyed in this paper. It shows how the different approaches and models presented in the next Section interact and what processes are involved in the performance evaluation activities.

The *component developer* is responsible for supplying parametrised performance models of software components. To get parametrised component performance models, the component developer can start from component specifications and estimate the missing values, such as resource demands and parameter dependencies (upper left in Fig. 4). The software performance engineering (SPE) methodology by Smith and Williams [83] gives many hints on how to estimate these values before implementation. The component developer can also start from an implemented component (lower left in Fig. 4), which can then be analysed with static code analysis methods (preliminary work in [48]) or executed in a testbed (more in Section 3.2.1).

The component developer puts the component performance models into a *repository*. Such a repository might be publicly accessible via the web or company internal. Besides the component performance models, it can for example also include component specifications, implementations, documentation, etc. The component developer might also use components from such a repository to build new composite components.

The *software architect* assembles component performance models from the repository for a planned architecture. This includes specifying the wiring of the components (assembly model), specifics of the component platform and hardware (deployment model), and the usage information, such as workload, input parameters, and internal state (runtime model). Section 3.1.3 lists approaches for benchmarking middleware platforms. The output of these approaches provides useful information for the deployment model. Section 3.1.1 and 3.1.2 list different approaches, which allow the specification of such architectural models.

A model transformation can then transform a complete architecture model (i.e., assembly, deployment, runtime) into a classical system-wide performance model. Notations for such performance models include (extended) queueing networks, stochastic Petri nets, and stochastic process algebras [1]. The models enable deriving performance metrics, such as response time, throughput, and resource utilisation. The software architect can compare these values to the performance requirements of the planned system.

If the prediction results show that the requirements cannot be fulfilled with the given design, the architecture model needs improvement in a design-space exploration step (top middle of Fig. 4). Smith et al. [83] list several common performance principles, patterns, and anti-patterns, which are useful to improve the design. The software architect can also use models of other components or alter the performance requirements, which however means renegotiation with the customer. Afterwards, the updated architecture model can again be transformed and its performance be predicted.

If the prediction results show that the design can fulfil the

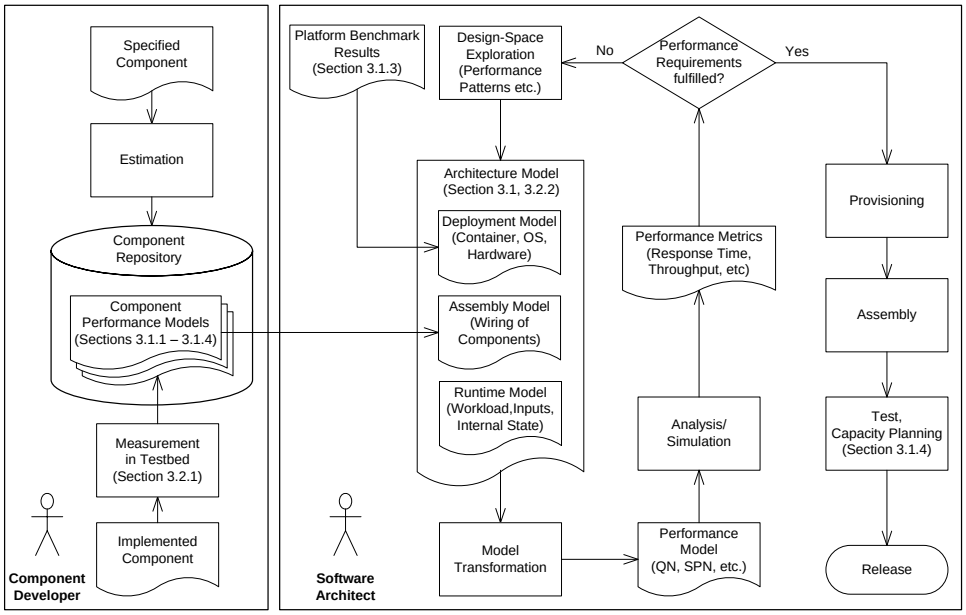

Figure 4: Process Model

requirements, the implementation of the system can start. During the provisioning phase, the software architect makes build or buy decisions for the components specified in the architecture model. Once all components are bought or implemented, the software architect can assemble them and then test the whole system. Besides testing functionality, the software architect can also measure the performance of the system to perform capacity planning (i.e., determining the maximal workload). Section 3.1.5 lists approaches for this task. Finally the software architect releases the complete system to the customers.

3. Performance Evaluation Methods

In the following, we provide short summaries of the performance evaluation methods for component-based software systems in the scope of this survey. We have grouped the approaches as depicted in Fig. 5

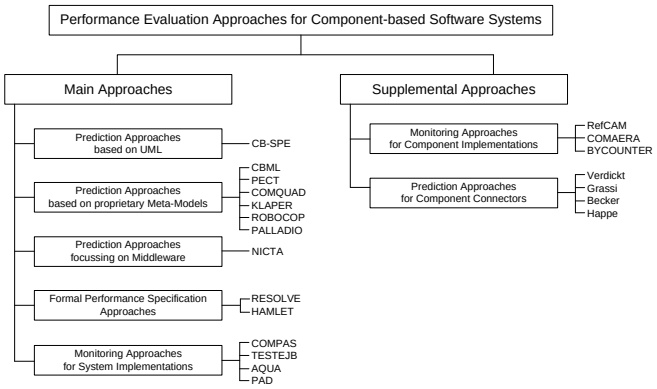

Figure 5: Overview of the Methods

Main approaches provide full performance evaluation processes, while supplemental approaches focus on specific aspects, such measuring individual components or modelling the performance properties of component connectors (i.e., artefacts binding components). The groups have been chosen based on the similarity of the approaches. Approaches within one group can be best compared against each other. There are groups where only one approach has been included in this survey, although there are multiple similar approaches reported in literature. We have excluded those approaches that do not exhibit tool support or rely on obsolete technologies, as described further below.

We do not strictly distinguish between model-based and measurement-based approaches as former surveys (e.g., [7]). Most methods built around UML or special meta-models involve some kind of measurements to determine the performance annotations in the models. On the other hand, most methods measuring implemented component-based systems feature some form of model (e.g., to predict changes or to analyse the measurement results). Therefore it is not appropriate to divide the approaches into model-based or measurement-based approaches.

For each approach, we describe its goals and context, its modelling language, available tools, documented case studies, and extensions. We postpone an evaluation of the approaches until Section 4.

3.1. Main Approaches

3.1.1. Prediction Approaches based on UML

Approaches in this group target performance predictions during design time for component-based software systems modelled with the Unified Modelling Language (UML) [73]. UML 2.0 has a notion of a software component as an extended

class. UML allows modelling component behaviour with sequence, activity, and collaboration diagrams. Component allocation can be described with deployment diagrams. While UML only supports functional specifications, its extensions mechanism (profiles, consisting of stereotypes, constraints, and tagged values) have been used by the OMG to allow modelling performance attributes such as timing values and workload parameters. The UML SPT profile [68] for UML 1.4 from 2002 has recently been replaced by the UML MARTE profile [72] for UML 2.1.

Early approaches in this group by Kähkipuro et al. [47] and Gomaa et al. [30] were proposed before the introduction of the UML SPT profile. They relied on proprietary extensions to UML and are therefore outdated. The approach by DiMarco et al. [22] uses the UML SPT profile, but has been excluded from this survey, because it features neither divided developer roles nor tool support.

(CB-SPE) [8]: The Component-Based Software Performance Engineering (CB-SPE) approach by Bertolino and Mirandola uses UML extended with the SPT profile as design model and queueing networks as analysis model. The modelling approach is divided into a component layer and an application layer. On the component layer, developers model the schedulable resource demands of individual performance services in dependence to environment parameters. There are no guidelines on how to obtain these values. The parametrisation does not involve service input or output parameters nor explicit required service calls. There is no support for analysing memory consumptions.

In the application layer, software architects pre-select components performance models and compose them into architecture models. They model the control flow through the architecture using sequence diagrams and have to anticipate the expected flow through the components, because the component specifications do not refer to component interaction. Workload attributes (e.g., user population) is added to the sequence diagrams, while the deployment environment is modelled using deployment diagrams.

The CB-SPE framework includes freely available modelling tools (ArgoUML) and performance solvers (RAQS), and includes a transformation tool to map the UML model to execution graphs and queueing networks. The solution of the queueing networks is analytical (using the parametric decomposition approach). An approach by Balsamo et al. [2] is planned as an extension to the CB-SPE framework and allows computing the upper bounds on throughput and response times before actually solving a queuing network.

3.1.2. Prediction Approaches based on proprietary Meta-Models

The approaches in this group aim at design time performance predictions. Instead of using the UML as the modelling language for component developers and software architects, these approaches feature proprietary meta-models.

(CBML) [96]: The Component-Based Modelling Language (CBML) by Wu and Woodside is an extension to layered queueing networks (LQN) [80]. LQNs model the behaviour and resource demands of software entities with so-called 'tasks'. Resource demands are specified as mean values of exponential distribution functions, but there is no support for memory consumption.

CBML extends tasks to model reusable software components. Therefore, it adds a so-called "slot" to a task, which contains a number of interfaces representing provided and required services of a component. Slots make CBML components replaceable with other components conforming to the same slot. Additionally they allow nesting components into composite components.

CBML components include placeholders for resources to be executed on and components to be connected to. Software architects create CBML bindings to connect these placeholders to other elements in their architectural LQN model. Bindings may contain a set of parameter values (e.g., to model a thread pool size or adjust resource demands). These do not refer to input or output parameters in interfaces and also cannot modify the control flow of the component. CBML supports synchronous and asynchronous communication among components as well as passive resources.

There is an XML schema to define CBML components, but there are no graphical editors for CBML models. A complete CBML specification of a software architecture can be directly processed by the LQN solvers with numerical techniques or simulation. The method has been customised for EJB systems by Xu et al. [98].

(PECT) [45, 89, 44]: The Prediction Enabled Component Technology (PECT) by Hissam, Wallnau, et al. features a reasoning framework for performance evaluation of component-based software architectures. It consists of an analysis theory (i.e., rate monotonic analysis for predicting schedulability of real-time tasks), generation of theory-specific models (i.e., component composition language, CCL), and the evaluation of the models.

CCL is used for architectural description. It supports synchronous and asynchronous communication with required services. CCL also allows to specify component behaviour with statecharts. Resource demands, which can be constants or distribution functions are attached to the CCL components using annotations. CCL supports composite components but not memory consumption. For analysis, tools transform CCL models into a so-called intermediate constructive model (ICM), which focusses on parts relevant for performance analysis and eases the implementation of further transformations.

PECT mainly targets analysing real-time properties of component-based embedded systems. From the ICM, tools generate models for rate monotonic analysis (RMA) or simulation using the MAST tool. There is a graphical modelling and analysis tool called PACC starter kit available [66]. The authors report on a large scale industrial case study in [43].

(COMQUAD) The project "Components with Quantitative properties and Adaptivity" (COMQUAD) developed a component container architecture capable of checking performance requirements on software components at runtime [27]. Multiple component implementations with different performance properties specified by so-called CQML+ descriptors shall be in-

stalled in the such a component container. The container then checks the performance requirements of clients against the performance specifications to select the components where to direct the client requests. The approach aims at EJB and Corba CCM systems and allows stream-based interfaces.

Meyerhöfer et al. [65] describe the component performance modelling language targeted by the project in detail. The response time of a component assembly is computed as the sum of the response times of all components. The approach discusses the issue of platform independent resource demands and suggests to use lookup tables for different platforms. The language introduces concepts to handle different kinds of required service calls as well as internal state dependencies (also see [63].

Two prototypical container implementations are available, one based on a C++ real-time container running on the real-time operating system DROPS [27], and another one based on a JBoss container running in a Java virtual machine. The approach targets checking performance properties at runtime and does not involve model building to analyse contention effects. No industrial case study has been reported so far.

(KLAPER) [32]: This method for performance prediction of component-based software systems by Grassi et al. includes a so-called kernel modelling language called KLAPER. The language is implemented in MOF and aims at easing the implementation of model transformations from different kinds of component system models (e.g., UML, OWL-S) into different kind of performance models (e.g., Markov chains, queueing networks). With KLAPER, it shall be possible to combine component performance models by different component developers based on different notations in a single prediction. It is not the goal that component developers model their components with KLAPER, but they shall implement model transformations from their own proprietary notations into KLAPER.

Therefore, the authors defined the language to be a minimal set of modelling constructs necessary for performance evaluation. Hardware resources as well as software components are modelled with the same constructs. There is no support for composite components. The language includes scheduled and limited resource demands as well as control flow. The authors implemented QVT transformations from KLAPER to EQNs and Markov chains. Because the language shall only be used by model transformations, there are no graphical modelling tools. The language is able to model dynamic software architectures where the connectors between components can change at runtime [34]. No industrial case study of KLAPER has been reported.

(ROBOCOP) [12]: This method adds a performance prediction framework on top of the ROBOCOP component system model [28] aiming at analysing component-based, embedded systems. ROBOCOP components have provided and required interfaces with multiple service signatures. Composite components are not supported. Additionally, ROBOCOP components contain a resource specification and a behavioural specification as well as the executable implementation of the component.

Component developers specify ROBOCOP components in a parametrised way. Software architects compose these specifications and instantiate parameters by the component developers. Scheduled resource demands of software components are constants in ROBOCOP, which also allows limited resource demands for semaphores and memory. The component specification allows only limited control flow (i.e., sequence and loops). There are no probabilistic attributes (e.g., transition probabilities, random variables) in the specification.

The so-called Real-Time Integration Environment (RTIE), which is implemented as a number of Eclipse plugins, supports the whole design and performance prediction process with ROBOCOP. It includes a component repository to store component specifications in XML and a graphical designer to model component assemblies as well as deployment environments. Using a preprocessor, RTIE transforms a complete architectural model into a task tree, which can then be simulated with different scheduling policies, such as rate monotonic scheduling. The results are response times to detect missed deadlines and processor utilisations to detect overloaded resources. The authors report on a design space exploration tool for the approach and an industrial case study involving a JPEG decoder [10].

(PALLADIO) [6]: This approach from the University of Karlsruhe is based on the Palladio Component Model (PCM) implemented with EMF. The approach enables component and architecture modelling and targets performance predictions for distributed systems. Component interfaces are first-class entities in the PCM and can be provided or required by multiple components. The interface syntax is based on CORBA IDL. The PCM supports building composite components.

The PCM is tailored for a strict separation of four developer roles, who participate in building model instances. *Component developers* model the performance properties of component services with annotated control flow graphs, which include resource demands and required service calls (so-called resource demanding service effect specifications, RDSEFF). They can parametrise RDSEFFs for input and output parameter values [53] as well as for the deployment platform [55]. Resource demands can be specified using general distribution functions. *Software architects* compose these component performance models into component assemblies. *System deployers* model the deployment environment as well as the allocation of components to hardware resources. Finally, *domain experts* model the usage profile (including workload, user flow, and input parameters).

The so-called PCM-Bench tool (implemented as a set of Eclipse plugins) allows independent graphical modelling for all four developer roles. Model transformations weave performance-relevant middleware features (e.g., connector protocols [4], message channel properties [39]) into the models. Further model transformations map the whole model into performance models (i.e., EQNs, LQNs), which are solved by simulation or numerical analysis. There is also support to generate code stubs and performance prototypes from the models. So far, the authors have reported no industrial case study.

3.1.3. Prediction Approaches with focus on Middleware

The following approaches emphasise the influence on the middleware on the performance of a component-based system.

Consequently, they measure and model the performance properties of middleware platforms, such as CORBA, Java EE, and .NET. The underlying assumptions of these approaches is that the business logic of the components itself has little impact on the overall performance of the system and thus does not require fine-granular modelling.

An early approach by Llado et al. [59] created an EQN model to describe the performance properties of an EJB 1.1 server. Cecchet et al. [13] benchmarked Java Enterprise application servers with an auction application called RUBiS and found that the most influencing factor on the performance for this application was the middleware. Denaro et al. [20] generated a prototype for a component-based design of an EJB system, but did not provide model building or tool support. Chen et al. [15] built a simple model to determine optimal thread pool sizes for Java application servers, which is part of the NICTA approach.

(NICTA) [58]: This performance prediction approach from a research group of NICTA (Gorton, Liu, et al.) targets server-side component technologies, such as EJB, .NET, and CORBA. It does not distinguish between a component developer and software architect role and assumes that the middleware has a significantly higher impact on performance than single software components. The authors present a queueing network model for component containers, which can be solved using Mean Value Analysis (MVA) [77].

To determine the parameters of the QN model, the authors analyse different architectural patterns for EJB applications. They model application services with activity diagrams based on patterns such as "container-managed persistence" and "find-by-non-primary-key". The activities in these diagrams refer to generic container services, such as "load from cache" or "activate / passivate".

The activity diagrams are further augmented with use-case specific information, such as the number of times a certain pattern is executed and the frequency of each transaction type. The resulting model still contains placeholders for services provided by the component container, which are used as platform-independent resource demands. To get these values, the authors have implemented a benchmarking application, which they execute on the target platform of a planned system. With this information the QN can be used to analyse the performance of the application under different workloads.

The authors report on a case study performed with a stock broker application and show sufficiently accurate prediction results. So far, the approach is specific for EJB applications. The approach does not include a seamless tool-chain, therefore the software architect needs to build the benchmarks and the QN manually. There are plans to build a capacity planning tool suite called Revel8or based on the approach [102].

3.1.4. Formal Performance Specification Approaches

Formal performance specification approaches for software components focus on describing the performance properties of components in a sound manner. These approaches target a fundamental theory of performance specification and do not deal with prediction or measurement frameworks.

(RESOLVE) [82]: Sitaraman et al. propose an extension dialect to the RESOLVE specification and implementation language for software components to express performance properties. The aim is to provide specifications of the execution time and memory consumption of component services in a refined big-O notation [51]. Assertions about the asymptotic form of execution time and memory consumption shall later enable formal verification. It is not intended to derive accurate response times or resource utilisations.

RESOLVE specifies the functionality of a component with a list of service signatures and a pre- and post-condition for each service. The components of RESOLVE do not declare required interfaces. The authors point out the limitations of classical big-O notations for generic software components with polymorphic data types. Therefore, they augment the functional RESOLVE specifications with an adapted big-O notation for execution time and memory consumption. These specifications are more refined, because they take the structure of input parameters into account. However, the specification does not distinguish between different processing resource (e.g., CPU and hard disk), does not include calls to required services, and also does not deal with passive resources.

The approach is not exhibited by tool support, but the authors demonstrate the feasibility on a simple example component.

(HAMLET) [36]: The approach by Hamlet et al. originates from the area of software testing. It was first proposed for reliability prediction of component-based systems [38], and later extended for performance prediction [35, 36]. The authors use a restricted component definition to reduce the complexity for creating a fundamental theory of software composition. A software component is a mathematical function with single integer value input. Component composition of two software components means that the first component sends all its outputs to the second component in a pipe-and-filter style. The method involves component specification by component developers and performance prediction by software architects.

Component developers specify subdomains for the input domain of their components, which partition the input domain into equivalence classes. They measure the execution time for the component for each subdomain and store the results in a repository. Software architects retrieve these specifications and the components. By executing the components with their desired usage profile, they can determine the call propagation of each component (i.e., which outputs will fall into the subdomains of connected components). With this information and the execution times measured by the component developers they can predict the performance of a complete system.

The approach does not consider concurrency, passive resources, or scheduling. However, it considers the influence of the internal state of components to performance in a restricted form. The approach is accompanied by a computer-aided design tool (CAD) for testing components to create performance specifications and calculating the overall performance from a set of composed components and an input profile. As the approach assumes a very restricted class of components, there is no validation on an industrial system.

3.1.5. Measurement Approaches for System Implementations

The following approaches assume that a complete component-based system has been implemented and can be tested. The goal is to find performance problems in the running system, identify performance bottlenecks, and adapt the system so that is able to fulfil certain performance requirements. An early approach by Yacoub [99] described systematic measurement of a component-based system, but provided no additional tool support.

(COMPAS) [67]: The COMponent Performance Assurance Solutions (COMPAS) framework by Mos et al. is a performance monitoring approach for J2EE systems. Components are EJBs in this approach. The approach consists of three parts: monitoring, modelling, and prediction. For *monitoring*, the authors use the Java Management Extensions (JMX) to augment a complete EJB application with proxy components for each EJB, which send timestamps for EJB life-cycle events (such as start-up or invocation) to a central dispatcher. Running the application yields performance measures from the proxy components, which can be visualised with a proof-of-concept graphical tool.

The approach then uses a *modelling* technique by generating UML models with SPT annotations from the measured performance indices. Users can then specify different workloads (e.g., number of simultaneous users, or inter-arrival rate) in the models to assess various scenarios. For performance *prediction* of the modelled scenarios, the approach suggests using existing simulation techniques, which are not part of the approach. Besides detecting bottlenecks in the system, prediction result should also be feed back into the instrumented application to focus the collection of timestamps on performance hotspot thereby reducing the overhead for monitoring.

(TestEJB) [65]: The TestEJB framework by Meyerhöfer targets performance monitoring of J2EE systems. It implements a profiling technique, which is application independent and more lightweight than available commercial J2EE profiling tools. Besides the execution times of individual EJBs, the framework can also determine call traces from single users through a whole architecture. This technique also works in the case of multiple concurrent users or multiple component instances.

The approach is based on the interceptor patterns and implemented as an extension to the JBoss application server. Interceptors log invocations of EJBs and augment calls with identifiers, so that it is possible to construct call graphs from execution traces. Measurement data from the interceptors is persisted to files asynchronously. TestEJB uses a bytecode instrumentation library to modify components at deployment time. The instrumentation process is completely transparent for the users. In a case study, TestEJB show a shows only a very small overhead for profiling in the range of microseconds.

The authors have extended the framework in [64] to also allow memory profiling of EJB applications. This approach relies on the Java Virtual Machine Profiler Interface (JVMPI) and records events for constructing and deconstructing EJBs. While this method allows tracing back memory consumption to individual components, it introduces a significant overhead in the range of seconds and therefore should only be used in a testing environment and not during production.

(AQUA) [21]: The automatic quality assurance (AQuA) framework by Diaconescu et al. builds on the foundations of the COMPAS framework but focusses on adapting a component-based application at runtime if performance problems occur. The main idea is that a system replaces a software component with performance problems with a functional equivalent one with different performance properties from a set of redundant components. The approach comprises of system monitoring, learning performance characteristics, anomaly detection, component evaluation, adaptation decision, and component activation.

For system monitoring, the JBoss application server is modified to collect timestamps from the execution of individual EJBs. All measurement data is stored to enable learning specific performance characteristics of each component. Furthermore, the approach involves monitoring the workload of a running application. Upon changing workloads, the approach may detect a violation of a response time requirement. This leads to evaluating redundant components for each software component to determine a component most fitted for the new workload. The hot deployment facility of JBoss is then responsible for swapping problematic component with other components. The approach has been demonstrated on Duke's bank account application from the J2EE tutorial.

(PAD) [74]: This approach by Parson et al. targets automatic detection of performance anti-patterns [83] in running component-based systems. It builds on the COMPAS framework and has implemented the performance anti-pattern detection (PAD) tool. The approach targets EJB systems and includes performance monitoring, reconstruction of a design model, and anti-pattern detection.

The *monitoring* relies on proxies in front of each EJB to collect timestamps and call sequences. The approach allows determining run-time paths through the application even if multiple clients interact with the application simultaneously. To overcome the needed redeployment of a running system, the approach plans to implement a byte code instrumentation tool, which shall dynamically instrument the application at runtime.

Recorded measurements and information from EJB deployment descriptors is used to *reconstruct a design model*. The reconstruction process involves clustering techniques and statistical analysis on the recorded run-time paths. The design model is based on a proprietary meta-model, which features components, methods, run-time paths, pools, and tracked objects.

Anti-pattern detection on the reconstructed design model relies on rules implemented with the JESS rule engine. The approach distinguishes between anti-patterns across or within run-time paths, inter-component relationship anti-patterns, anti-patterns related to component communication patterns, and data tracking anti-patterns. The authors applied the approach and the PAD tool on Duke's bank account application (and found for example not required stateful session beans) and IBM's Workplace application (and found for example simultaneous exposure of local and remote interfaces of a bean).

3.2. Supplemental Approaches

3.2.1. Measurement Approaches for Component Implementations

The goal of measurement approaches for single software component implementations is to derive parametrised performance specifications via multiple measurements. They aim at factoring out the usage profile, required service, and the deployment platform from the performance specification, so that it can be used in different contexts. These approaches usually include a testbed to execute the components and a statistical evaluation framework to derive functions for the resource demands from measurement data via regressions.

(RefCAM) [93]: Woodside et al. present an approach to determine resource function for software components. Resource functions express the resource demands (e.g., CPU demand, memory demand) of a component in dependency to input parameter values and execution environments. RefCAM comprises of a test harness to run measurement experiments with single software components and a statistical module to determine resource functions from the measurement data.

The test harness allows executing a software component repetitively with a manually defined test plan, which defines rules for varying the arguments of functions provided by the component. This can include varying the size of messages sent by the component, the number of calls to required services, and changes in the execution environments. The framework also allows to recompile software components to change variables in the implementation.

RefCAM records CPU demands and applies function fitting techniques, such as linear regression or multi-variate adaptive regression splines to express the parametric dependencies of the resource functions. The results are stored in a repository, which shall support recording multiple resource functions of the same component for different execution environments. RefCAM has been implemented in the ObjectTime CASE tools.

The approach does not deal with the issue of providing the required services of a software component to execute it. Instead it assumes that the required services are available for testing. Furthermore, it specifies CPU demand in seconds, therefore not parametrising the resource functions for the underlying deployment platform.

(COMAERA) [63]: This approach by Meyerhöfer targets the platform-independent measurement of Java components. It assumes that the execution time of a software component can be divided into a number of primitive elements (so-called atoms). Measuring the number of executions of each atom yields a platform-independent CPU demand specification for a software components. Software architects shall combine these specifications with benchmarking results from the desired target platform, which provide the execution time for each atom. The combination leads to a prediction of the execution time of the software component on the target platform without actually running it on the platform.

The authors analyse different possibilities to choose the unit for the atoms, ranging from individual bytecode instruction to instruction groups. Instruction groups can for example be based on homogeneous execution times of specific instructions or even be specific for certain components.

Because the resolution of typical Java performance counters is too coarse to measure the execution time of individual instructions, the approach uses an indirect method to determine their execution time. It first executes a number of microbenchmarks based on different algorithms with known execution numbers of instructions or instruction groups on the target platform and measures the overall execution time. Then, it constructs a linear equation system based on the execution numbers and measured times and solves it for the execution times of individual instructions. Because there is no general solution for the equation system, the approach determines a heuristic solution minimising the error.

The authors present a case study with more than 20 test programs and make performance predictions for different platforms. The error of predictions versus measurements is in some cases less than 10 percent, but in other cases (e.g., with included native code) higher than 100 percent. The approach assumes absence of just-in-time compilation. It furthermore does not deal with calls to required services or different input parameters.

(ByCounter) [55]: This approach by Kuperberg et al. aims at constructing parametrised, behavioural performance models from Java components, which can be used for performance prediction for different usage profiles and execution environments. The approach comprises (i) reconstructing platform-independent component performance models, (ii) running a benchmark application on the desired target platform to determine platform dependent performance properties, and (iii) performance prediction by combining the results from the first two steps. Therefore, a newly designed component-based system does not need to be tested on the target environment, instead a more cost-efficient, model-based prediction can be made.

The approach uses the number of bytecode instructions executed by a component service as a platform-independent unit for CPU demand. It distinguishes between multiple classes of bytecode instructions as well as Java API calls. Using monitored input parameters from running instances of a software component, the approach executes a component repeatedly in a testbed and records the execution times. It assumes that required services of a component are available for testing. Then it runs genetic algorithms on the measurement data to determine parametric dependencies and reconstruct a behavioural model.

The user has to run the benchmark application on the target platform to determine the execution times for bytecode instructions and Java API calls. Multiplying the execution times from the benchmark with the parametrised resource demands in the reconstructed behavioural model yields the expected execution time of a component service on the new target platform. The authors report on a case study, where they predicted the performance of file sharing application in different execution environments and for different usage profiles. The ByCounter approach shall be ultimately integrated into the Palladio approach.

3.2.2. Prediction Approaches with focus on Component Connectors

These approaches assume an existing component description language and focus on modelling and measuring the performance impact of the connections between components. These connections can be implemented with different middleware techniques. They can for example include performance overheads for encryption or calling naming services in distributed applications. These overheads can significantly alter the overall performance of a component-based system, as the connections might be realised with slow or overloaded network connections. All of the following methods weave the performance overhead of the middleware into a platform-specific model using model transformations according to the MDA approach.

Verdickt2005 [88]: This approach introduces a model transformation framework to include the performance overhead of middleware layers into a component-based system model. Following MDA guidelines, a model transformation maps a platform-independent model based on UML activity, deployment, and collaboration diagrams, into a platform-specific model using a repository of middleware models also based on UML. All UML models are annotated with performance estimates using the UML SPT profile.

The approach then suggests to transform the resulting annotated UML model into an LQN, which can be analysed for the desired performance metrics. In a case study, the authors demonstrate how to include the performance impact of the CORBA Object Request Broker (ORB) into the models. This performance overhead results for example from naming or security services. In the case study, the performance annotations for the middleware models have been determined via measurements from a prototype application. These performance annotations are not parametrised for different network connection speeds or message sizes.

Grassi2006 [31]: The authors propose a connector refinement transformation from high-level UML 2.0 architecture models into the KLAPER modelling language. KLAPER models can be transformed into Markov chains or queueing networks for performance analysis. The approach assumes the availability of a UML stereotype to label component connectors. For example, a connector could realise a static or dynamic synchronous client/server interaction.

The authors propose to build a library of parametric connector behaviours models in KLAPER for each of the stereotypes. The behaviour of the connectors may for example involve marshalling functionality or calling naming services. For a specific deployment platform, the parametrisation of these models can be solved and they can be weaved into the KLAPER model of the whole architecture replacing the component connectors. The transformation is demonstrated through an example, which relies on the QVT Relations [70] transformation language. Ultimately, the refined KLAPER model is transformed in a queueing network, which can then be used for performance evaluation.

Becker2008 [4]: This approach presents a configurable model transformation, which adds the performance overhead of component connector realisations (e.g., SOAP or RMI) to a component-based performance model. The underlying performance description language is the Palladio Component Model (PCM).

A software architect can parametrise the model transformation for different message sizes and choose multiple middleware services, such as authentication and encryption, to make the performance model even more accurate. The performance annotations for the middleware model are determined using a benchmark application on the desired target platform. They are parametrised over different usage profiles, but dependent on a specific hardware.

The author reports on a simple case study involving a client-server system deployed on Sun's Glassfish Java EE server. Predictions based on simulation of the resulting PCM model differ less than 5 percent from measurements.

Happe2008 [39]: This approach proposes a model transformation framework, which extends Palladio with the overhead for using a Message-oriented Middleware (MOM). The authors analyse the performance impact of different messaging patterns, such as point-to-point or publish-subscriber channels as well as different message or connection pool sizes.

The approach requires software architects to run a benchmark application on the target system. It determines parametrised performance annotations for all MOM configurations supported by the framework. The results are stored in a repository. Different software architects can later configure a model transformation to add specifically configured middleware models (e.g., for a specific messaging pattern or a specific message size) for the repository to their overall performance model. The model transformation adds components to the performance models, which exhibit the performance overhead of the MOM.

In a case study involving the SPECjms2007 benchmark application, the authors demonstrate a prediction error for response times of the models below 20 percent for multiple configurations of the transformation.

3.2.3. Miscellaneous Approaches

The following approaches provide interesting contributions to the performance evaluation of component-based software systems. They are, however, no complete prediction or measurement approaches and have different goals than the approaches described so far.

Woodside et al. [95] introduced the concept of performance completions. These are model elements, which are added to a functional model of a software system to enable performance evaluations. Such elements may include model annotations, software components, environment infrastructure models, deployment models, communication pattern models, or design refinements. Developers shall store such model completions into repositories, so that different software architects can extend their platform-independent design models to platform-specific models. This concept is for example used by the performance modelling approaches for component connectors in Section 3.2.2.

Eskenazi et al. [25] extended the so-called APPEAR method for performance prediction of component-based software systems. The method involves creating a prediction model for individual component operations, creating annotated control flow graphs for single component services, and creating and simulating a full application model, which may include concurrent execution of component services. However, the approach neither features a meta-model nor tool support and is therefore difficult to repeat.

Menasce et al. [62] propose the concept of QoS-aware software components, which are able to negotiate performance properties with their clients at runtime. Upon receiving a request for service with a certain QoS requirement, these components build a QN model based on previously monitored performance properties at runtime and evaluate whether they are able to fulfil the request with the required QoS.

Reussner et al. [78] developed the concept of parametric performance contracts for software components. This approach features a parameterised performance specification of a software component, which can be used to calculate the provided performance of a component in case the performance of required services is known. The approach shall help software architects at design time to identify components that deliver suitable performance properties given a specific deployment environment.

4. Evaluation

This section first discusses the general features of the surveyed main approaches (Section 4.1.1), before classifying them according to the expressiveness of their modelling languages (Section 4.1.2). The Sections 4.2.1- 4.2.5 additionally analyse the benefits and drawbacks of the different groups of approaches surveyed in Section 3.

4.1. Feature Discussion

4.1.1. General Features

Table 1 summarises the general features of the surveyed main approaches. The following elaborates on each of the features.

The *target domain* of the methods is either generic (e.g., for formal specification methods), embedded real-time systems, or distributed systems. The domain of embedded systems involves real-time operating systems with for example rate monotonic scheduling. It is the goal of methods for embedded systems to predict the violation of hard real-time deadlines or to evaluate whether a given design is able to run on hardware devices with limited processing power and memory. Software components in this domain are typically small and their behaviour is easy to describe. This is reflected in the component specification languages of the relevant approaches (i.e., PECT and ROBOCOP).

The domain of distributed systems involves general purpose operating systems with sophisticated scheduling algorithms running on multicore processors. Besides the operating system, distributed systems may rely on middleware and virtual machines, which all may impact the perceived performance of a component-based system. Rather than assessing the possible violation of deadlines, it is the goal of performance prediction methods for these systems to provide a probabilistic answer whether a given design can be expected to fulfill certain service level agreements. Violating these agreements is not as critical as violating the deadlines of real-time systems. Thus, these approaches work with approximations and estimations. Software components in this domain can be considerably large, so that their behaviour cannot be described in detail and has to be abstracted to construct an analysable model at all (cf. CBML, PALLADIO).

While the target domains induce different viewpoints for the performance prediction, the concept of a software component described by provided and required interfaces is not different. It should be possible to use the same description techniques for embedded and distributed systems, which then can be analysed with different prediction techniques.

Each approach uses a different component description language, also analysed in detail in Section 4.1.2). Some approaches use annotated UML models (e.g., CB-SPE, NICTA), other approaches use domain-specific meta-models (e.g., CBML, CCL, ROBOCOP, PALLADIO). For other approaches (e.g., COMPAS, AQUA, NICTA, PAD) the component description is given by the specification of EJBs. No methods specific for .NET or CORBA CCM specification are known. Lau [56] surveys further software components models, which however usually do not provide special support for performance predictions. A standard notation for component performance specification has not been achieved today, because the domain of performance modelling is still not understood well enough [19].

The performance evaluation methods use *model transformations* to generate performance models from the component system models with deployment and usage profile models. Besides KLAPER, none of the methods uses a standardised model transformation framework, such as QVT [71]. Existing transformations are usually implemented in an ad-hoc manner using C or Java code.

Most of the surveyed methods rely on (extended) queueing networks as the *performance model* and incorporate existing performance solvers for numerical solution or simulation. This is possible because the fully composed and annotated system model of a component-based system can be treated like a monolithic system for the actual performance analysis. Numerical solutions for QNs are applicable only for a restricted class of models, because they are based on assumptions such as exponentially distributed execution times or missing synchronisation mechanisms. Simulation techniques are able to handle a larger class of models. However, they are usually more difficult to construct and the execution of the simulation may require a high amount of time if prediction results with high statistical significance are desired [46].

Ideally, tools should *feed back* the prediction results into the design model. This allows the software architect to work without knowledge of the underlying performance theory. Except for ROBOCOP and PAD none of the approaches offers any automated support into this direction. Therefore, software architects need to interpret the results by themselves. The per-

Name	RESOLVE	COMPAS	HAMLET	CB-SPE	COMQUAD	CBML	PECT	KLAPER	ROBOCOP	AQUA	NICTA	PALLADIO	PAD
Target Domain	Generic	Distributed Systems	Generic	Distributed Systems	Distributed Systems	Distributed Systems	Embedded Systems	Distributed Systems	Embedded Systems	Distributed Systems	Distributed Systems	Distributed Systems	Distributed Systems
Component Design Model	RESOLVE	EJB 2.0 + annotated UML diagrams	Simple Mathematical Functions	Annotated UML diagrams (UML SPT)	CQML+	CBML	CCL	UML+SPT, OWL-S, KLAPER	ROBOCOP	EJB 2.0	EJB 2.0 + UML activity diagrams	PCM	EJB 3.0 + RDMM
Model Transformation	n/a	n/a	n/a	ad-hoc (C-code, XMI Input/Output)	n/a	n/a	ATL, ad-hoc	QVT	ad-hoc (RTIE tool)	n/a	n/a	ad-hoc (Java)	n/a
Performance Model	n/a	not specified	Simple Algebra	Execution Graph + QN	Execution Graphs	LQN	RMA, QN, RTQT, MAST	Markov Chain, EQN	Task Tree	n/a	QN	EQN, LQN	RDMM
Performance Model Solver	n/a	not specified	Analytical	Analytical (MVA)	Analytical (no queueing)	Analytical (approximative), Simulation	Analytical, Simulation	Simulation	Simulation	n/a	Analytical (MVA)	Analytical + Simulation (LQN), Simulation (EQN)	JESS Rule Engine + PAD tool
Prediction Feedback into Design Model	n/a	n/a	n/a	planned	n/a	n/a	n/a	planned	Areas of Interests Visualizer	n/a	n/a	planned	Anti-pattern detection from monitoring data
Automated Design Space Exploration	n/a	n/a	n/a	n/a	n/a	yes (Performance Booster tool)	n/a	n/a	yes (incl. Pareto analysis performance vs. costs)	yes (exchanging components at runtime)	n/a	planned	n/a
Tool Support	n/a	Monitoring (COMPAS framework)	Modelling, Analysis (CAD tool)	Modelling, Analysis (CB-SPE Tool Suite)	Monitoring	Modelling (Jlqndef), Analysis (LQNS), Simulation (LQSIM)	Modelling, Simulation (PACC Starter KIT)	Model Trans-formation (KLAPER QVT)	Modelling, Simluation (RTIE)	Monitoring, Adaptation (AQUA_J2EE)	Benchmarking (for J2EE app servers)	Modelling, Analysis, Simluation (PCM-Bench)	Monitoring, Anti-pattern detection (PAD tool)
Case Study	n/a	Sun Petstore, IBM Trade3	Artificial Components	Software Retrieval System	Xvid Video Codec (EJB), Generic EJB Application	Abstract 3-tier architecture	Large Real Time Control System	Generic Client/Server system	MPEG4 decoder	Duke's Bank Account	Online Stock Broker based on EJB	Media Store system	Duke's Bank Account, IBM Workplace Application
Maturity and Applicability in Industry	Low (tools+case study missing)	Medium (case study exists, but tools outdated)	Low (tools prototypical, restricted component model)	Medium (case study exists, tools outdated)	Medium	Medium (tools available, but improvable usability)	High (industrial case study available)	Medium (tools still work in progress)	High (matured tools + case study)	Low (tool unavailable)	Medium (industrial case study, but tools not finished)	High (matured tools based on up-to-date technology)	High (multiple case studies available)

Table 1: General Features of the Performance Evaluation Methods

formance annotations of the UML SPT and MARTE profile offer support for displaying performance results in performance models. However, this possibility has not been exploited by any of the models.

Design space exploration deals with automatic generation and evaluation of new design alternatives in case performance problems are found in a design. Software architects shall get hints on how to improve their design based on documented performance patterns and anti-patterns [83]. CBML identifies performance problems and suggests to improve specific hardware resources or the execution time of specific software resources [97]. ROBOCOP can evaluate a set of given design alternatives and perform a Pareto analysis that compares costs for each alternatives with the expected performance [10]. More research is needed for example to let the approaches determine appropriate middleware configurations and deal with black-box component specifications.

The *tool support* of the approaches ranges from performance profiling testbeds for individual components, graphical modelling tools for architecture specification, model transformation, to performance analysis or simulation. However, with few exceptions [43, 60], the tools have only been used by their authors themselves. They are usually merely research prototypes, which are sparsely tested and documented. Often they are hardly robust against improper use and hard to use for non-experts.

The *case studies* carried out to demonstrate the benefits of the approaches usually involve small self-designed systems (e.g., CBML, CB-SPE, PALLADIO) or rely on standard industry examples (e.g., from the Java EE tutorials).

4.1.2. Language Expressiveness

Table 2 depicts the expressiveness of the component performance specification languages employed by the surveyed approaches according to the feature diagram in Section 2.4. The following discusses these features in detail.

For specifying scheduled *resource demands* most methods simply use platform dependent timing values. They could be adapted to different platform by multiplying them with a factor for a slower or faster resource (e.g., proposed by CBML and CB-SPE). However, the feasibility of this approach has not been demonstrated by any of the methods. Methods with platform independent scheduled resource demands are still under development. An extension to PALLADIO [55] proposes using individual Java bytecode instructions as platform independent resource demand. COMQUAD suggests using different bytecode instruction classes [63]. These approaches are however limited if native code is embedded in a component. A demonstration of such a platform-independent specification for a large-scale component-based system is still missing.

Several methods allow distribution functions to specify resource demands. This is especially useful for large software components, where it is hard to describe the resource demand as a constant value. General distribution functions usually required simulation techniques to evaluate the models, however the results are more expressive than constant values or exponential distributions. For embedded systems there is additionally special interest in best-case, average-case, and worst-case

Name	RESOLVE	COMPAS	HAMLET	CB-SPE	COMQUAD	CBML	PECT	KLAPER	ROBOCOP	AQUA	NICTA	PALLADIO	PAD
Scheduled Resource Demands	Extended Landau Notation	Timing Value, Constant	Timing Value, Constant	Timing Value, Constant	CPU demand in bytecode instr., distributions	Timing Value, Distribution Functions	Timing Value, Distribution Function	Timing Values, Constant	Timing Values, Constant	Timing Value, Constant	Timing Values, Constant	Timing Values, Distribution Functions	Timing Values, Constant
Limited Resource Demands	n/a	n/a	n/a	n/a (only specification, no analysis)	n/a	Semaphores	Semaphores	Semaphores	Memory	n/a	n/a	Semaphores	Memory, Pools
Required Service Calls	n/a	Synch	Synch	Synch, Asynch, (specified by SA)	Sync, Asynch	Synch, Asynch	Synch, Asynch	Synch, Asynch	Synch, Asynch	Synch	Synch	Synch	Synch
Behaviour	n/a	n/a (Control flow specified by SA)	n/a (Control flow specified by SA)	n/a (Control flow specified by SA)	Sequence, Alternative, Loop	Sequence, Alternative, Loop, Fork	Sequence, Alternative, Loop, Fork	Sequence, Alternative, Loop, Fork	Sequence, Loop	unknown	Sequence, Alternative, Loop	Sequence, Alternative, Loop, Fork	Sequence, Alternative, Loop
Parametric Dependencies	Res. Demand	n/a	Loop, Alternative, Res. Demand	n/a (not for service parameters)	Res. Demand	n/a (not for service parameters)	n/a	informal	Loop, Res. Demand, Req. Serv. Call	unknown	n/a (not for service parameters)	Loop, Alternative, Res. Demand, Req. Serv. Call	n/a
Internal State	n/a	n/a	yes (treated as additional input)	n/a	yes (Petri-nets)	n/a	yes (using statecharts, perf. impact unknown)	n/a	n/a (Comp. Parameter)	unknown	n/a	n/a (Comp. Parameter)	n/a

Table 2: Component Performance Model Expressiveness Comparison

performance metrics. For individual components these values can be determined using methods and tools surveyed in [90].

Limited resource demands (e.g., for acquiring/releasing semaphores or thread pool instances) are often critical for the performance of component-based systems. Several methods allow modelling the acquisition and release of semaphores. Chen et al. [15] specifically deal with determining appropriate thread pool sizes for EJB application servers. Only a few approaches allow the analysis of memory consumption, which is an important performance metric if the system will be deployed on small embedded devices or if garbage collection may impact performance significantly.

Most of the methods support analysing synchronous as well as asynchronous *calls to required services*. Building performance models for component connectors, which might involve modelling network connections and protocols, is described in Section 3.2.2.

Control flow differs across the surveyed methods. RESOLVE does not model any control flow in the component performance specification and assumes components, which do not require other components. COMPAS, HAMLET, and CB-SPE force the software architect to model the control flow through the architecture and do not provide component internal behaviour specifications. This approach might be infeasible, if a software architect assembles black-box components, for which the control flow propagation is not known.

PECT and ROBOCOP model control flow on a detailed level, however ROBOCOP allows no alternatives or forks. PALLADIO proposes to include only control into the component performance specification if required services are involved. The control flow between two subsequent required service calls is abstracted, and a single resource demand models the behaviour between the calls. This grey-box view can be extracted from black-box components via measurements. In general, different abstraction levels for component behaviour are required for different systems. For small components in embedded systems it might be appropriate to express the behaviour of every statement with a single resource demand, while large components in distributed systems may require abstractions to keep the model analysable.

Parameter dependencies can be attached to behavioural nodes (e.g., loop counts, branch conditions), to required service calls (e.g., the input parameter for such calls), or on resource demands (e.g., CPU demand depending on input parameter value). ROBOCOP and PALLADIO allow parameter dependencies for behaviour, required service calls, and resource demands (cf. [11, 54]). ROBOPCOP uses constants to specify parameter values, whereas PALLADIO additionally supports random variables to specify parameter values for large user groups. Some methods (e.g., CB-SPE, CBML, NICTA) include parameter dependencies in their specification, which are not directly linked to parameters declared in component interfaces. CBML for example allows to change the capacity of limited resources and the number of replications of a component using variables.

Only a few methods deal with modelling the *internal state* of a component at runtime, although state can cause sever changes to a component's performance. COMQUAD suggests using Petri nets to model internal state, whereas PECT uses UML state charts. HAMLET suggests a restricted model, where the internal state of a component consists of a single floating point value. More research is needed to adequately model the internal state for performance prediction.

4.2. Critical Reflection

The following section provides a critical view on the surveyed approaches (also see Tab. 3).

4.2.1. Prediction Approaches based on UML

Early approaches in this group ([47, 30]) relied on proprietary UML extensions, while all later approaches use the UML SPT profile for model annotation introduced in 2002. Future approaches in this group will likely make use of UML MARTE profile from 2008, which succeeds the SPT profile. There are many more UML performance prediction approaches [1], which target object-oriented or monolithic systems instead of component-based systems. Thus, these approaches are out of

	Benefits	Drawbacks
Prediction Methods based on UML	- based on standards - developer friendly notation	-bad tool support -conversion of existing models needed
Prediction Methods with proprietary Meta Model	- good tool support - sound component notions	-proprietary notations
Prediction Methods focussing on Middleware	- accurate predictions - close to existing systems	- test system required for measurements - tool support still limited
Formal Specification Methods	- sound component notions	- limited analysis and prediction frameworks
Measurement Methods for System Implementations	- accurate performance analysis - easy to apply	- not applicable during design

Table 3: Evaluation of the Methods

scope for this paper. While it is possible to describe and analyse component-based systems with these approaches, it would require special mechanisms and high developer discipline to create reusable (i.e., parametrised) models independently by component developer and software architects.

The main *benefit* of UML-based approaches is their compliance to OMG standards, which could enable using any existing UML CASE tools to specify the models. Bridging the semantic gap between design models familiar to developers and performance models with sometimes complicated mathematical solution techniques, is highly beneficial to make performance analysis more practitioner friendly. Hiding the complexity of the performance models by using model transformations and results feedback could increase the acceptance of performance analysis techniques in regular software development processes. Furthermore, UML-based approaches enable developers to reuse existing UML models, thereby lowering the effort for performance analysis.

However, several *drawbacks* have limited UML-based approaches for performance evaluation of component-based systems. The concept of reusable software component has only subsequently been introduced into UML 2.0 and is still subject to discussion. It is hardly possible for component developers and software architects to work independently of each other using UML.

Existing UML performance profiles do not include special concepts to parametrise component performance specifications for different usage profiles or resource environments. While the SPT profile allows variable specifications in performance annotations, these usually refer to global variables in the model defined in a configuration file. Their link to input or output parameters of a component interface would only be implicit, which hinders different component developers and software architects to communicate with each other. The UML SPT profile allows parametrisation to different execution environments by specifying processing rates of individual hardware nodes. Such a model does not account for aspects of the middleware underlying a component-based system.

The tool support for the existing methods in this group is still weak, as there are at most sparsely tested research prototypes. A reason for this might be the difficulty to implement model transformations with UML as the source model, because of the ambiguities of this semi-formal language. A second factor are the missing transformation frameworks, as the QVT standard [71] for UML model transformation has only been adopted recently.

So far there is low interest from industry in these methods. There are hardly any industrial case studies reported. Many companies use UML only as a documentation tool. It would require a major effort to adjust existing models so that they have the necessary formality to be processed by model transformation tools. The complexity of the latest UML versions (more than 1200 pages) and the UML performance profiles (more than 400 pages for the UML MARTE profile) induces a steep learning curve on the specification language demotivating regular developers.

4.2.2. Prediction Methods based on proprietary Meta-Models

The methods in this category are based on their own meta-models and do not rely on the UML. A recent trend is the use of the Eclipse Modeling Framework to implement such meta models. With the corresponding Graphical Editing Framework (GEF) and the Graphical Modeling Framework (GMF), it is possible to semi-automatically generate graphical editors for these meta-models. This allows researchers to quickly build modelling tools to assess their models and connect performance solvers.

The *benefit* of these approaches is thus the often good tool support with graphical editors, model transformations to known performance models, numerical analysis tools, and simulators. Proprietary meta-models allow researchers to create new abstractions of component-based systems, which might capture the performance-critical properties more accurately than a UML model. These models usually use a more strict component concept than the UML, where a software component is an extended form of a class. The semantics of the models are often more formally defined than for UML. Because of their special purpose character, such modelling languages are at the same time more restricted than the UML and might be easier to learn for developers. Languages such as ROBOCOP or PALLADIO additionally feature a role concept, which provides specific models for component developers and software architects and therefore enables these roles to work independently.

As a *drawback*, these methods are not standard-conforming, which results in several challenges. Developers first have to learn a new language and then re-formulate existing UML models in the proprietary language, which might not be straightforward, because of different abstraction levels. Existing commercial CASE tools cannot be reused for modelling, because they only support specific UML versions.

4.2.3. Prediction Methods with Focus on Middleware

The *benefit* of these methods is their high prediction accuracy and their easy applicability in industry. These methods provide a good mixture of modelling techniques for component

business logic and measurement techniques for component infrastructure (e.g., middleware, operating system). For systems where component middleware has a high influence on the overall performance, these methods are best suited. Because of their focus to a specific middleware implementation, these methods can be quickly applied by practitioners.

Unfortunately, these methods require set-up, running, and benchmarking a specific middleware environment, which can be costly. Their portability is often limited as they are usually restricted for a very specific middleware version. They are often outdated if a new version of a middleware appears. The component specification language of such approaches is usually simple, as the middleware is assumed to be the determining factor for performance. Therefore, these methods might be difficult to apply on component-based systems with a complex business logic. As another drawback, there is still limited tool support for these methods, so that developers are forced to create their own middleware benchmark test applications.

4.2.4. Formal Specification Methods

These methods are theoretically sound and do not aim at the latest component technologies from industry. They focus on accurate component specification languages with appropriate abstraction levels. They offer a good comprehension of component and composition concepts. However, these methods still have weak tool support, as there are hardly any model transformations and performance solvers implemented based on them. A validation of the introduced concepts based on the analysis of real, large-scale systems is missing. Industry applicability is a long term goal for these methods.

4.2.5. Measurement Methods for System Implementations

These methods offer special measurement and analysis facilities for a completely implemented and running component-based system. They provide more functionality than available middleware profilers (e.g., [23]) and often help to understand the influences on performance in a given system better than pure measurement tools.

As a *benefit*, these methods can offer accurate measurement results for a running system. These methods assist developers in finding performance bottlenecks in a given system or to identify implemented performance anti-pattern. They are useful for capacity planning and re-design of legacy systems.

As a *drawback*, these methods are only applicable after system implementation and not during system design. They are complimentary to the prediction methods and should be applied to validate model-based predictions. Furthermore, these methods are usually tied to a specific middleware version and are not portable to other middleware. The models produced by these methods often only have a low degree of parametrisation (e.g., for different usage profiles). Thus, predictions based on these models are less flexible (e.g., the required services cannot be changed). Some of the methods (e.g., PAD) construct restricted prediction model, which do not derive quantitative performance metrics, such as overall response times or throughputs.

5. Future Directions

This survey has revealed many open issues and recommendations for future work in performance evaluation of component-based software systems:

Model Expressiveness: As shown by the survey and discussed in the former section, a common performance modelling language for component-based systems has not been achieved yet. Component performance can be modelled on different abstraction levels. The question is which detail to include into the performance models because of its impact on timing and which detail to abstract because of its limited impact. Too much detail might make the models intractable for numerical analysis models as well as for simulations.

The goal is to create an abstraction of a software component, which allows accurate performance prediction results. However, in some scenarios a specific parameter might have a large performance impact, while in other scenarios it is negligible. Thus, a common abstract performance model might not be possible. Instead, for large software components it could be beneficial to have multiple performance models from which software architects can then select depending on the kind of scenario they want to analyse.

As for the features of a performance modelling languages described in Section 2.4 most existing methods do not support modelling internal state and parameter dependencies well. More research and experiments into this direction are necessary. To make the performance model of the middleware more accurate, different model completions (i.e., model extensions for specific aspects) [95] should be designed (examples in [88, 4, 39]).

Runtime and Dynamic Architectures: The surveyed methods support modelling the runtime life-cycle stage of software components only in a limited way. They include the workload and the usage profile of the component-based system at runtime. However, nowadays distributed or grid-based systems may involve dynamic reconfiguration of the architecture at runtime [26]. There are only initial approaches supporting performance prediction for dynamic architectures (e.g., KLAPER [33]).

Furthermore, online performance monitoring at runtime can be combined with modelling techniques to react on changing usage profiles and deployment environments. In this direction, prediction techniques could help to identify the boundaries of such changes, so that a component-based system can still work according to certain service level agreements. Another interesting direction is performance negotiation between different components at runtime, which has been proposed by Menasce et al. [62].

Domain-Specific Approaches: While some of the surveyed approaches are tailored for a specific target system domain (i.e., embedded or distributed systems) or for a specific technical domain (e.g., EJB systems), other domain-specific methods and models could improve performance evaluation of component-based systems. For certain business domains (e.g., accounting systems, retail systems, etc.), patterns for the software components could be identified so that special domain-

specific modelling languages and prediction methods can be designed. This would potentially enable rapid creation of prediction models and therefore reduce the costs for a model-driven performance analysis.

Support for Service-oriented Systems: The concepts developed for performance evaluation of component-based systems can potentially be applied also for service-oriented systems, which often rely on similar technologies. In fact, many service-oriented systems are built from deployed software components. Service providers procure the resource environment for the components, therefore performance predictions do not need to parametrise over the resource environment as it cannot easily be altered by the software architect.

Performance prediction methods for component-based systems could be helpful in establishing service level agreements. Due to the distributed nature of services, performance modelling must put special emphasis on the performance of network protocols, such as SOAP or REST. Future service-oriented systems (e.g., cloud computing applications) may be hosted in large data centers introducing a high degree of scalability, but also a high complexity for performance models and predictions. New abstractions levels beyond component-based performance modelling need to be created to deal with such systems.

Expressive Feedback: Both performance measurement and prediction methods should give the software architect detailed feedback on the expected performance of a system and how to improve it. Most of the current approaches simply provide measured or predicted performance metrics, such as response times, throughput, and resource utilisation. They require the software architect to manually draw conclusions from these metrics. This is especially difficult if the performance metrics stem from a performance model (e.g., a QN or Petri net) and cannot be directly linked to components and connectors in the design model.

Some initial approaches for giving software architects more sophisticated feedback from the performance prediction have been reported [18, 97]. In the future, these approaches should account for the characteristics of component-based systems, where for example the internals of third-party components cannot be changed. Prediction methods should assist software architects in finding performance anti-patterns [83, 74] and implementing performance patterns. Trade-off analysis methods also with other QoS-attributes (e.g., reliability, availability, security, etc.) are required.

Feedback from performance prediction methods could also be systematically collected into a performance knowledge basis [94]. This could allow identifying new performance anti-patterns and help to create a set of common performance solutions for reoccurring performance problems.

Hybrid Solution Techniques: Hybrid performance solvers combine numerical analysis methods with simulation approaches. None of the surveyed methods makes use of a hybrid performance solution techniques, although multiple methods offer analytical as well as simulation solvers. Hybrid methods could be a solution for the state-space explosion of large performance models and allow efficient and accurate performance analyses. How to exploit such techniques in a component-based setting is still unknown. It could be possible to segment a large model into smaller parts and solve individual parts using analytical techniques and using the results to create a simulation model or vice versa. Verdickt et al. [87] integrate a network simulator into an analytical performance solver for LQNs. Such techniques could improve the accuracy of predictions for network intensive component-based systems.

Reverse Engineering: While there are sophisticated metamodels and performance prediction techniques available, in industry a crucial issue for performance analysis is often how to build a performance model for a legacy system. With the strict time-constraints in industry, an automated solution based on static code analysis or profiling is desirable. In a component-based setting, a software architect could be interested in the performance of a legacy or third party component. Some methods for component performance measurement have been surveyed in Section 3.2.1, but still need more research to be usable in industry.

For a test-bed for component implementations there are three major challenges: (i) support for parametrising over different usage profiles, (ii) support for parametrising over required services, and (iii) support for parametrising over different platforms.

For the first challenge, it is impossible for non-trivial software component to profile them for their whole input domain. Therefore, suitable abstractions for the input domain based on equivalence classes need to be found [37, 63].

For the second challenge, it is necessary to generate stubs for the required services of a component, so that it can be tested independent from other components. However the return values from required services also belong to the input domain of the component. Therefore, as in (i) it is necessary to vary the return values created by the generated stubs to cover the whole input domain.

For the third challenge, it is necessary to find a suitable platform-independent performance metric. For Java-based systems, counting the number of bytecode instructions of a component implementation has been proposed [65, 55]. As an example, Binder et al. [9] have implemented a profiler for Java applications, which counts the number of bytecode instructions computed and the number of objects allocated. For embedded system components running on small devices without a complicated middleware and operating system, it might be possible to count the required CPU cycles [11]. More experience and case studies involving industrial systems are needed in this direction.

The use of Kalman filters [101] to estimate model parameters not directly measurable from black-box component implementations could be an interesting direction for future research.

Model Libraries: As software components, parametrised component performance specifications shall be reused by multiple software architects, so that the extra effort for creating a parametrised model pays off. Software architects shall use model repositories or model libraries to retrieve existing performance models to be included into their models as completions [95] and to store new performance models [94, 18].

Such repositories could also used to systematically store measurement results so that performance degradation could be

reported or that initial models are later refined with more accurate measurements. These measurements could also be benchmarking results for certain middleware products, such application servers. Although Internet-based component marketplaces have not succeeded so far, company-internal component repositories enhanced with reusable performance models could allow rapid performance predictions for software product lines.

Empirical Studies: While a high automation degree is desired, most performance evaluation methods require manual input by component developers and software architects. This includes estimating specific performance properties, building models using graphical editing tools, performing statistical analyses to get parametrised performance models, and interpreting the results of performance solvers. It is still unknown what degree and what quality of human input is necessary to achieve accurate performance evaluations.

Empirical studies are useful to analyse the human aspects of performance evaluation methods. This includes comprehension of the proposed models and usage of the accompanying tools. Experiments with multiple students can analyse whether the necessary inputs can be provided by average developers and whether the tools produce results suitable for non-experts. While initial studies have been performed [60], replication and analysis of other aspects is desirable.

Empirical studies involving cost-benefit analysis are still missing. It is still unclear how much costs early design time performance predictions save compared to late development cycle performance fixing. Such studies require to execute the same project twice with both approaches, which is very expensive.

Empirical studies are hard to design, require major effort to conduct, and the results are often debatable [92]. However, such studies have the important side effect that they require the performance modelling and prediction tools to be used by third parties. Thus, it is possible to reveal deficits of existing research prototype tools and make the existing approaches more mature and robust so that they become easier applicable in industrial settings.

Increase Technology Transfer: Most of the presented methods have not reached a level of industrial maturity. They have often been tested on small example systems many times designed by the authors themselves. Some approaches use example implementations of middleware vendors, such as Duke's Bank Account from Sun to validate their approaches. However, there are hardly attempts to build reusable component performance models for these systems.

The Common Component Modelling Example (CoCoME) [76] is an effort by several research groups to design a standard system for evaluation. A Java-based implementation of the system is available so that model-based performance prediction from the design can be compared with actual measurements. Nevertheless, the system is a research prototype.

Hissam et al. [43] reported on a large-scale case study, where the authors applied the PECT method on an embedded system with more than one million lines of code. Other methods should also be applied on larger industrial system to check their scalability and validate the chosen abstraction levels. Report of such case studies would help practitioners to use the methods, thereby avoiding performance firefighting [81].

Teaching the performance predictions methods to practitioners is another important factor for increasing the technology transfer from academia to industry. Matured methods should not only report on real-life case studies, but also be accompanied with consistent and up-to-date documentation and teaching guides. Many methods are merely documented through research papers or PhD thesis and lack extensive tool documentation.

Performability There are many extra-functional properties besides performance, such as reliability, availability, security, safety, maintainability, usability etc. For component-based systems for example researchers have proposed many reliability prediction approaches [29]. It is desirable to combine the evaluation methods for other extra-functional properties with performance evaluation methods as some of these properties also depend on execution times (e.g., energy consumption). Different methods could benefit from each other, as they may propose behavioural abstraction levels suitable for multiple extra-functional properties.

6. Conclusions

We have surveyed the state-of-the-art in research of performance evaluation methods for component-based software systems. The survey classified the approaches according to the expressiveness of their performance modelling language and critically evaluated the benefits and drawbacks.

The area of performance evaluations for component-based software engineering has significantly matured over the last decade. Several issues have been understood as good engineering practise and should influence the creation of new approaches. A mixed approach, where individual components as well as the deployment platform are measured and the application architecture and the usage profile are modelled, is advantageous to deal with the complexity of the deployment platform while at the same time enabling early life-cycle performance predictions. The necessary parametrised performance modelling language for software components has become more clear. Including benchmarking results for component connectors and middleware features into application models using model completions exploits the benefits of model-driven development for performance evaluation.

Our survey benefits both researchers and practitioners. Researchers can orient themselves with the proposed classification scheme and assess new approaches in the future. The listed future directions for research help identifying areas with open issues. Practitioners gain an overview of the performance evaluation methods proposed in research. They can select methods according to their specific situation and thus increase the technology transfer from research to practise.

A generic approach applicable on all kinds of component-based systems may not be achievable. Instead, domain-specific approaches could decrease the effort for creating performance models.

Acknowledgements: The author would like to thank the members of the SDQ research group at the University of Karlsruhe, especially Anne Martens, Klaus Krogmann, Michael Kuperberg, Steffen Becker, Jens Happe, and Ralf Reussner for their valuable review comments. This work is supported by the EU-project Q-ImPrESS (grant FP7-215013).

References

[1] Simonetta Balsamo, Antinisca DiMarco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *IEEE Trans. Softw. Eng.*, 30(5):295–310, May 2004.

[2] Simonetta Balsamo, Moreno Marzolla, and Raffaela Mirandola. Efficient performance models in component-based software engineering. In *EUROMICRO '06: Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 64–71, Washington, DC, USA, 2006. IEEE Computer Society.

[3] F. Bause and F. Kritzinger. *Stochastic Petri Nets - An Introduction to the Theory*. Vieweg Verlag, 2nd edition, 2002.

[4] Steffen Becker. *Coupled Model Transformations for QoS Enabled Component-Based Software Design*. PhD thesis, University of Oldenburg, Germany, January 2008.

[5] Steffen Becker, Lars Grunske, Raffaela Mirandola, and Sven Overhage. Performance Prediction of Component-Based Systems: A Survey from an Engineering Perspective. In Ralf Reussner, Judith Stafford, and Clemens Szyperski, editors, *Architecting Systems with Trustworthy Components*, volume 3938 of *LNCS*, pages 169–192. Springer, 2006.

[6] Steffen Becker, Heiko Koziolek, and Ralf Reussner. Model-based performance prediction with the palladio component model. In *Proc. 6th Int. Workshop on Software and Performance (WOSP'07)*, pages 54–65, New York, NY, USA, 2007. ACM.

[7] Steffen Becker and Ralf Reussner. The Impact of Software Component Adaptation on Quality of Service Properties. *L'objet*, 12(1):105–125, 2006.

[8] Antonia Bertolino and Raffaela Mirandola. Cb-spe tool: Putting component-based performance engineering into practice. In *Proc. 7th Int. Symposium on Component-based Software Engineering (CBSE'04)*, number 3054 in LNCS, pages 233–248. Springer, May 2004.

[9] Walter Binder, Jarle Hulaas, Philippe Moret, and Alex Villazon. Platform-independent profiling in a virtual execution environment. *Software Practice and Experience*, 39(1):47–79, January 2009.

[10] Egor Bondarev, Michel R. V. Chaudron, and Erwin A. de Kock. Exploring performance trade-offs of a JPEG decoder using the deepcompass framework. In *Proc. of the 6th International Workshop on Software and performance (WOSP '07)*, pages 153–163, New York, NY, USA, 2007. ACM.

[11] Egor Bondarev, Peter de With, Michel Chaudron, and Johan Musken. Modelling of Input-Parameter Dependency for Performance Predictions of Component-Based Embedded Systems. In *Proc. of the 31th EUROMICRO Conference (EUROMICRO'05)*, 2005.

[12] Egor Bondarev, Johan Muskens, Peter de With, Michel Chaudron, and Johan Lukkien. Predicting Real-Time Properties of Component Assemblies: A Scenario-Simulation Approach. In *Proc. 30th EUROMICRO Conf. (EUROMICRO'04)*, pages 40–47, Washington, DC, USA, 2004. IEEE Computer Society.

[13] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. Performance and scalability of ejb applications. In *Proc. 17th ACM SIGPLAN Conf. on Object-oriented programming, systems, languages, and applications (OOPSLA'02)*, pages 246–261, New York, NY, USA, 2002. ACM.

[14] J. Cheesman and J. Daniels. *UML Components: A Simple Process for Specifying Component-based Software Systems*. Addison-Wesley, 2001.

[15] Shiping Chen, Yan Liu, Ian Gorton, and Anna Liu. Performance prediction of component-based applications. *J. Syst. Softw.*, 74(1):35–43, 2005.

[16] Microsoft Corp. The COM homepage. `http://www.microsoft.com/com/`. last retrieved 2008-01-13.

[17] Microsoft Corporation. *Improving .NET Application Performance and Scalability (Patterns & Practices)*. Microsoft Press, 2004.

[18] V. Cortellessa and L. Frittella. A framework for automated generation of architectural feedback from software performance analysis. In *Proc. 4th European Performance Engineering Workshop (EPEW'07)*, volume 4748 of *LNCS*, pages 171–185. Springer, September 2007.

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

[20] Giovanni Denaro, Andrea Polini, and Wolfgang Emmerich. Early performance testing of distributed software applications. In *Proc. 4th Int. Workshop on Software and Performance (WOSP'04)*, pages 94–103, New York, NY, USA, 2004. ACM.

[21] Ada Diaconescu and John Murphy. Automating the performance management of component-based enterprise systems through the use of redundancy. In *Proc. 20th Int. Conf. on Automated Software Engineering (ASE'05)*, pages 44–53, New York, NY, USA, 2005. ACM.

[22] Antinisca DiMarco and Paola Inverardi. Compositional generation of software architecture performance qn models. In *Proc. 4th Working IEEE/IFIP Conference on Software Architecture (WISCA'04)*, pages 37–46. IEEE, June 2004.

[23] ej technologies. JProfiler. `http://www.ej-technologies.com`, 2008. last retrieved 2009-01-07.

[24] Sun Microsystems Corp., The Enterprise Java Beans homepage. `http://java.sun.com/products/ejb/`, 2007. last retrieved 2008-01-13.

[25] Evgeni Eskenazi, Alexandre Fioukov, and Dieter Hammer. Performance Prediction for Component Compositions. In *Proc. 7th International Symposium on Component-based Software Engineering (CBSE'04)*, volume 3054 of *LNCS*. Springer, 2004.

[26] Rich Friedrich and Jerome Rolia. Next generation data centers: trends and implications. In *WOSP '07: Proceedings of the 6th international workshop on Software and performance*, pages 1–2, New York, NY, USA, 2007. ACM.

[27] Steffen Göbel, Christoph Pohl, Simone Rö;ttger, and Steffen Zschaler. The comquad component model: enabling dynamic selection of implementations by weaving non-functional aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 74–82, New York, NY, USA, 2004. ACM Press.

[28] Jean Gelissen. Robocop: Robust open component based software architecture. `http://www.hitech-projects.com/euprojects/robocop/deliverables.htm`. last retrieved 2008-01-13.

[29] Swapna S. Gokhale. Architecture-based software reliability analysis: Overview and limitations. *IEEE Trans. on Dependable and Secure Computing*, 4(1):32–40, January-March 2007.

[30] Hassan Gomaa and Daniel A. Menasce. Performance engineering of component-based distributed software systems. In *Performance Engineering, State of the Art and Current Trends*, pages 40–55, London, UK, 2001. Springer-Verlag.

[31] V. Grassi, R. Mirandola, and A. Sabetta. A Model Transformation Approach for the Early Performance and Reliability Analysis of Component-Based Systems. In *Proc. 9th Int. Symposium on Component-Based Software Engineering (CBSE'06)*, volume 4063 of *LNCS*, pages 270–284. Springer, June 2006.

[32] Vincenzo Grassi, Raffaela Mirandola, and Antonino Sabetta. From design to analysis models: a kernel language for performance and reliability analysis of component-based systems. In *Proc. 5th International Workshop on Software and Performance (WOSP '05)*, pages 25–36, New York, NY, USA, 2005. ACM Press.

[33] Vincenzo Grassi, Raffaela Mirandola, and Antonino Sabetta. A Model-Driven Approach to Performability Analysis of Dynamically Reconfigurable Component-Based Systems. In *Proc. 6th Workshop on Software and Performance (WOSP'07)*, February 2007.

[34] Vincenzo Grassi, Raffaela Mirandola, and Antonino Sabetta. Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach. *Journal on Systems and Software*, 80(4):528–558, 2007.

[35] Dick Hamlet. Software component composition: subdomain-based testing-theory foundation. *Journal on Software Testing, Verification and Reliability*, 17:243–269, 2007.

[36] Dick Hamlet. Tools and Experiments Supporting a Testing-based Theory of Component Composition. *ACM Trans. on Softw. Eng. Methodol-*

ogy, To Appear, 2009.

[37] Dick Hamlet, Dave Mason, and Denise Woit. *Properties of Software Systems Synthesized from Components*, volume 1 of *Series on Component-Based Software Development*, chapter Component-Based Software Development: Case Studies, pages 129–159. World Scientific Publishing Company, March 2004.

[38] Dick Hamlet, David Mason, and Denise Woit. Theory of software reliability based on components. In *Proc. 23rd International Conference on Software Engeneering (ICSE'01)*, pages 361–370, Los Alamitos, California, May12–19 2001. IEEE Computer Society.

[39] Jens Happe, Holger Friedrichs, Steffen Becker, and Ralf Reussner. A Configurable Performance Completion for Message-Oriented Middleware. In *Proc. 7th International Workshop on Software and Performance (WOSP'08)*. ACM Sigsoft, June 2008. To Appear.

[40] M. Harkema, B. M. M. Gijsen, R. D. van der Mei, and L. J. M. Nieuwenhuis. Performance comparison of middleware threading strategies. In *Proc. of International Symposium on Performance Evaluation of Computer and Communication Systems (SPECTS'04)*, 2004.

[41] George T. Heineman and William T. Councill, editors. *Component-based software engineering: putting the pieces together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[42] Holger Hermanns, Ulrich Herzog, and Joost-Pieter Katoen. Process algebra for performance evaluation. *Theor. Comput. Sci.*, 274(1-2):43–87, 2002.

[43] S. Hissam, G. Moreno, D. Plakosh, I. Savo, and M. Stemarczyk. Predicting the behavior of a highly configurable component based real-time system. In *Proc. Euromicro Conf. on Real-Time Systems 2008 (ECRTS'08)*, pages 57–68, July 2008.

[44] Scott Hissam, Mark Klein, John Lehoczky, Paulo Merson, Gabriel Moreno, and Kurt Wallnau. Performance property theory for predictable assembly from certifiable components (pacc). Technical report, Software Engineering Institute, Carnegie Mellon University, 2004.

[45] Scott A. Hissam, Gabriel A. Moreno, Judith A. Stafford, and Kurt C. Wallnau. Packaging Predictable Assembly. In *Proc. IFIP/ACM Working Conference on Component Deployment (CD'02)*, pages 108–124, London, UK, 2002. Springer-Verlag.

[46] Raj Jain. *The Art of Computer Systems Performance Analysis : Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1991.

[47] Pekka Kahkipuro. UML-Based Performance Modeling Framework for Component-Based Distributed Systems. In *Performance Engineering, State of the Art and Current Trends*, pages 167–184, London, UK, 2001. Springer-Verlag.

[48] Thomas Kappler, Heiko Koziolek, Klaus Krogmann, and Ralf Reussner. Towards Automatic Construction of Reusable Prediction Models for Component-Based Performance Engineering. In *Proc. Software Engineering 2008 (SE'08)*, volume 121 of *LNI*. GI, February 2008.

[49] Rick Kazman, Len Bass, Mike Webb, and Gregory Abowd. SAAM: a method for analyzing the properties of software architectures. In *Proc. 16th International Conference on Software Engineering (ICSE '94)*, pages 81–90, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[50] Rick Kazman, M. Klein, and Paul Clements. ATAM: Method for Architecture Evaluation. Technical Report CMU/SEI-2000-TR-004, Carnegie Mellon University, Software Engineering Institute, 2000.

[51] Donald E. Knuth. *The Art of Computer Programming*, volume Volume 1: Fundamental Algorithms. Addison-Wesley Professional, 3rd edition, 1997.

[52] Samuel Kounev. Performance Modeling and Evaluation of Distributed Component-Based Systems Using Queueing Petri Nets. *IEEE Trans. Softw. Eng.*, 32(7):486–502, July 2006.

[53] Heiko Koziolek. *Parameter Dependencies for Reusable Performance Specifications of Software Components*. PhD thesis, University of Oldenburg, Germany, March 2008.

[54] Heiko Koziolek, Steffen Becker, and Jens Happe. Predicting the Performance of Component-based Software Architectures with different Usage Profiles. In *Proc. 3rd International Conference on the Quality of Software Architectures (QoSA'07)*, volume 4880 of *LNCS*, pages 145–163. Springer, Juli 2007.

[55] Michael Kuperberg, Klaus Krogmann, and Ralf Reussner. Performance Prediction for Black-Box Components using Reengineered Parametric Behaviour Models. In *Proceedings of the 11th International Symposium on Component Based Software Engineering (CBSE 2008), Karlsruhe, Germany, 14th-17th October 2008*, volume 5282 of *LNCS*, pages 48–63. Springer, October 2008.

[56] Kung-Kiu Lau and Zheng Wang. Software component models. *IEEE Transactions on Software Engineering*, 33(10):709–724, October 2007.

[57] E.D. Lazowska, J. Zahorjan, G.S. Graham, and K.C. Sevcik. *Quantitative System Performance*. Prentice Hall, 1984.

[58] Yan Liu, Alan Fekete, and Ian Gorton. Design-level performance prediction of component-based applications. *IEEE Trans. Softw. Eng.*, 31(11):928–941, November 2005. Member-Yan Liu and Member-Alan Fekete and Member-Ian Gorton.

[59] Catalina M. Llado and Peter G. Harrison. Performance evaluation of an enterprise JavaBean server implementation. In *Proc. 2nd Int. Workshop on Software and Performance (WOSP'00)*, pages 180–188, New York, NY, USA, 2000. ACM.

[60] Anne Martens, Steffen Becker, Heiko Koziolek, and Ralf Reussner. An Empirical Investigation of the Effort of Creating Reusable Models for Performance Prediction. In *Proc. 11th Int. Symposium on Component-based Software Engineerint (CBSE"08)*, volume 5282 of *LNCS*, pages 16–31. Springer, October 2008.

[61] M.D. McIlroy, JM Buxton, P. Naur, and B. Randell. Mass-Produced Software Components. *Software Engineering Concepts and Techniques (NATO Science Committee)*, 1:88–98, 1968.

[62] D. A. Menasce, V. A. F. Almeida, and L. W. Dowdy. *Performance by Design*. Prentice Hall, 2004.

[63] Marcus Meyerhöfer. *Messung und Verwaltung von Software-Komponenten für die Performance-Vorhersage*. PhD thesis, University of Erlangen-Nuernberg, 2007.

[64] Marcus Meyerhöfer and Frank Lauterwald. Towards platform-independent component measurement. In *Proceedings of the 10th Workshop on Component-Oriented Programming (WCOP2005)*, 2005.

[65] Marcus Meyerhöfer and Christoph Neumann. TESTEJB - A Measurement Framework for EJBs. In *Proceedings of the 7th International Symposium on Component-Based Software Engineering (CBSE7)*, 2004.

[66] Gabriel Moreno and Paulo Merson. Model-driven performance analysis. In *Proc. 4th Int. Conf. on the Quality of Software Architecture (QoSA'08)*, volume 5281 of *LNCS*, pages 135–151. Springer, 2008.

[67] Adrian Mos and John Murphy. A framework for performance monitoring, modelling and prediction of component oriented distributed systems. In *WOSP '02: Proceedings of the 3rd international workshop on Software and performance*, pages 235–236, New York, NY, USA, 2002. ACM.

[68] Object Management Group (OMG). UML Profile for Schedulability, Performance and Time. `http://www.omg.org/cgi-bin/doc?formal/2005-01-02`, 2005. last retrieved 2008-01-13.

[69] Object Management Group (OMG). Corba component model, v4.0 (formal/2006-04-01). `http://www.omg.org/technology/documents/formal/components.htm`, 2006. last retrieved 2008-01-13.

[70] Object Management Group (OMG). Metaobject facility (MOF). `http://www.omg.org/mof/`, 2006. last retrieved 2008-01-13.

[71] Object Management Group (OMG). MOF QVT final adopted specification (ptc/05-11-01). `http://www.omg.org/cgi-bin/apps/doc?ptc/05-11-01.pdf`, 2006. last retrieved 2008-01-13.

[72] Object Management Group (OMG). UML Profile for MARTE, Beta 1. `http://www.omg.org/cgi-bin/doc?ptc/2007-08-04`, August 2007. last retrieved 2008-01-13.

[73] Object Management Group (OMG). Unified modeling language: Infrastructure version 2.1.1. `http://www.omg.org/cgi-bin/doc?formal/07-02-06`, February 2007. last retrieved 2008-01-13.

[74] Trevor Parsons and John Murphy. Detecting Performance Antipatterns in Component Based Enterprise Systems. *Journal of Object Technology*, 7(3):55–90, March-April 2008.

[75] Erik Putrycz, Murray Woodside, and Xiuping Wu. Performance techniques for cots systems. *IEEE Softw.*, 22(4):36–44, 2005.

[76] Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and Frantisek Plasil, editors. *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *LNCS*. Springer, 2008.

[77] M. Reiser and S. S. Lavenberg. Mean-value analysis of closed multichain queuing networks. *J. ACM*, 27(2):313–322, 1980.

[78] Ralf H. Reussner, Steffen Becker, and Viktoria Firus. Component composition with parametric contracts. In *Tagungsband der Net.ObjectDays 2004*, pages 155–169, 2004.

[79] Ralf H. Reussner, Iman H. Poernomo, and Heinz 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, 2003.

[80] J. A. Rolia and K. C. Sevcik. The method of layers. *IEEE Trans. Softw. Eng.*, 21(8):689–700, 1995.

[81] Jayshankar Sankarasetty, Kevin Mobley, Libby Foster, Tad Hammer, and Terri Calderone. Software performance in the real world: personal lessons from the performance trauma team. In *WOSP '07: Proceedings of the 6th international workshop on Software and performance*, pages 201–208, New York, NY, USA, 2007. ACM.

[82] Murali Sitaraman, Greg Kuczycki, Joan Krone, William F. Ogden, and A.L.N. Reddy. Performance specification of software components. In *Proc. of SSR '01*, 2001.

[83] Connie U. Smith. *Performance Solutions: A Practical Guide To Creating Responsive, Scalable Software*. Addison-Wesley, 2002.

[84] T. Stahl, M. Voelter, and K. Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[85] Clemens Szyperski, Daniel Gruntz, and Stephan Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2002.

[86] Microsoft ACE Team. *Performance Testing Microsoft .NET Web Applications*. Microsoft Press, 2002.

[87] T. Verdickt, B. Dhoedt, F. De Turck, and P. Demeester. Hybrid Performance Modeling Approach for Network Intensive Distributed Software. In *Proc. 6th International Workshop on Software and Performance (WOSP'07)*, ACM Sigsoft Notes, pages 189–200, February 2007.

[88] Tom Verdickt, Bart Dhoedt, Frank Gielen, and Piet Demeester. Automatic inclusion of middleware performance attributes into architectural uml software models. *IEEE Transactions on Software Engineering*, 31(8):695–771, 2005.

[89] Kurt Wallnau and James Ivers. Snapshot of CCL: A Language for Predictable Assembly. Technical report, Software Engineering Institute, Carnegie Mellon University, 2003.

[90] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.

[91] Lloyd G. Williams and Connie U. Smith. Making the business case for software performance engineering. In *Proceedings of CMG*, 2003. last retrieved 2008-01-13.

[92] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering: an Introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[93] C. Murray Woodside, Vidar Vetland, Marc Courtois, and Stefan Bayarov. Resource function capture for performance aspects of software components and sub-systems. In *Performance Engineering, State of the Art and Current Trends*, pages 239–256, London, UK, 2001. Springer-Verlag.

[94] Murray Woodside, Greg Franks, and Dorina Petriu. The Future of Software Performance Engineering. In *Future of Software Engineering (FOSE '07)*, pages 171–187, Los Alamitos, CA, USA, May 2007. IEEE Computer Society.

[95] Murray Woodside, Dorin Petriu, and Khalid Siddiqui. Performance-related completions for software specifications. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 22–32, New York, NY, USA, 2002. ACM Press.

[96] Xiuping Wu and Murray Woodside. Performance Modeling from Software Components. In *Proc. 4th International Workshop on Software and Performance (WOSP'04)*, volume 29, pages 290–301, New York, NY, USA, 2004. ACM Press.

[97] Jing Xu. Rule-based automatic software performance diagnosis and improvement. In *WOSP '08: Proceedings of the 7th international workshop on Software and performance*, pages 1–12, New York, NY, USA, 2008. ACM.

[98] Jing Xu, Alexandre Oufimtsev, Murray Woodside, and Liam Murphy. Performance modeling and prediction of enterprise javabeans with layered queuing network templates. *SIGSOFT Softw. Eng. Notes*, 31(2):5, 2006.

[99] Sherif M. Yacoub. Performance analysis of component-based applications. In *SPLC 2: Proceedings of the Second International Conference on Software Product Lines*, volume 2379 of *LNCS*, pages 299–315, London, UK, 2002. Springer-Verlag.

[100] Peter Zadrozny, Philip Aston, and Ted Osborne. *J2EE Performance Testing*. A-Press, 2003.

[101] Tao Zheng, Murray Woodside, and Marin Litoiu. Performance model estimation and tracking using optimal filters. *IEEE Trans. on Softw. Eng.*, 34(3):391–406, May/June 2008.

[102] Liming Zhu, Yan Liu, Ngoc Bao Bui, and Ian Gorton. Revel8or: Model Driven Capacity Planning Tool Suite. In *Proc. 29th Int. Conf. on Software Engineering (ICSE'07)*, pages 797–800, Washington, DC, USA, 2007. IEEE Computer Society.