State Dependence in Performance Evaluation of Component-Based Software Systems

Lucia Kapova[⋆], Barbora Buhnova[†], Anne Martens[⋆], Jens Happe[‡] and Ralf H. Reussner[⋆,‡]
[⋆]Universität Karlsruhe (TH), 76131 Karlsruhe, Germany
Email: {kapova, martens, reussner}@ipd.uka.de
[†]Masaryk University, 60200 Brno, Czech Republic
Email: buhnova@fi.muni.cz
[‡]Forschungszentrum Informatik (FZI), 76131 Karlsruhe, Germany
Email: {reussner, jhappe}@fzi.de

ABSTRACT

Integrating rising variability of software systems in performance prediction models is crucial to allow widespread industrial use of performance prediction. One of such variabilities is the dependency of system performance on the context and history-dependent internal state of the system (or its components). The questions that rise for current prediction models are (i) how to include the state properties in a prediction model, and (ii) how to balance the expressiveness and complexity of created models.

Only a few performance prediction approaches deal with modelling states in component-based systems. Currently, there is neither a consensus in the definition, nor in the method to include the state in prediction models. For these reasons, we have conducted a state-of-the-art survey of existing approaches addressing their expressiveness to model stateful components. Based on the results, we introduce a classification scheme and present the state-defining and state-dependent model parameters. We extend the Palladio Component Model (PCM), a model-based performance prediction approach, with state-modelling capabilities, and study the performance impact of modelled state. A practical influences of the internal state on software performance is evaluated on a realistic case study.

Categories and Subject Descriptors: C.4 [Computer Systems Organization]: Performance of Systems – *performance attributes*; D.2.2 [Software Engineering]:Design Tools and Techniques
General Terms: Design, Performance

1. INTRODUCTION

During the last years, many approaches dealing with performance prediction and measurement were introduced. In the area of Component-Based Software Engineering (CBSE), systems are build out of reusable black-box components (implementing sets of services) interconnected to a component architecture. Specialised component performance prediction and measurement approaches introduce modelling languages with the aim to understand the performance (i.e. response time, throughput, resource utilisation) of a full architecture based on code-specific performance properties of individual components. It is generally accepted that performance is a pervasive quality of software systems. Everything affects it, from the software itself to all underlying layers, such as operating system, middleware, hardware, communication networks, etc. [26]. Moreover, the difficulty of understanding system performance comes from the propagation of the effects of these factors throughout system control flow, including the influence of the usage profile and history-dependent information defining system state. While the influence of usage profile on system control flow and subsequent performance has been studied and is commonly understood [15], not much attention has been paid to the influence of system stateful information. When speaking about a state, we mean a context or history-dependent information remembered inside a component or system, and used to coordinate system behaviour. State of a component or system can origin from its initialisation or previous executions, and can be changed at different stages of system life-cycle, including system initialization, deployment or runtime. Currently, there is no consensus in the definition and method to model stateful information in component-based systems and its performance impact, which limits the accuracy of existing performance models [17, 16].

1.1 Motivating Example

Consider a messaging system, implementing the Java Message Service standard [12], and supporting transaction for messages. The transactions guarantee that all messages are delivered to all receivers in the order they have been send. To achieve such a behaviour, Sun's JMS implementation MessageQueue 4.1 [1] waits for all incoming messages of a transaction and, then, delivers them sequentially. Figure 1 shows the measured delivery times for a series of transactions with 1000 messages each (the sender initiates a new transaction (as part of a session), passes 1000 messages to the MOM, and finally, commits the transaction). All messages arrive within the first 0.4 seconds and are delivered sequentially within the next second. This behaviour leads to delivery times of 0.4 seconds at minimum. The delivery

WOSP/SIPEW'10, January 28–30, 2010, San Jose, California, USA.

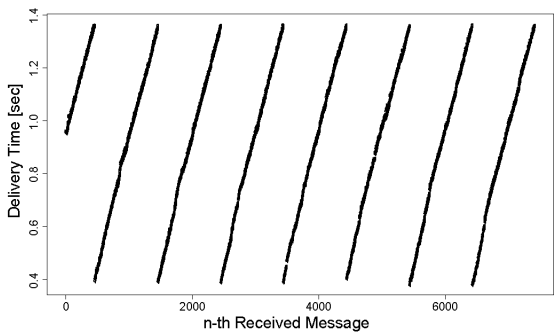

Figure 1: Time series of a transaction with 1000 messages per transaction set.

times grow linearly until the transaction is completed. In this example, the position of a message in the transaction set determines its delivery time. Thus, the measured delivery times are *not* independent and identically distributed but strongly depend on the number (and size) of messages that have already been sent. As a consequence, we need to keep track of the messages that are part of a transaction. Additionally, the periodical utilisation of resources (e.g., CPU) influences performance. To model such a behaviour, we need a notion of state as part of our performance model.

1.2 Challenges of Stateful Analysis

The question that rises for current performance models is how to include the software application properties identified above in a performance model, and how to build more accurate and expressive models of stateful component-based systems. In this respect, we can identify four main issues.

- **State definition:** The property of statefullness can be identified in various artifacts of component-based systems, varying over several system life-cycle stages. Existing literature lacks the localization of state-holding information identifiable in component-based systems [3, 16, 26], and their classification into a transparent set of categories. Available surveys consider the capability to model state only partially or not at all.

- **Performance impact:** The benefits of state modelling include increased expressive power of the models and higher accuracy of predictions. It is however not well studied, as observed by a number of authors [16, 4, 26, 16], what is the increase of prediction accuracy achieved by state modelling, especially in comparison to the increased effort for modelling and analysis. A discussion on how the existing performance-driven models deal with the interpretation and analysis of stateful prediction models is elaborated in section 3.1.

- **Prediction difficulty:** The balance between expressiveness (state modelling) and complexity (model size increase) is a challenging research question. Only when it is understood what costs need to be paid for the increase in prediction accuracy, we can competently decide on the suitable abstraction of state modelling (to what extent we aim to include stateful information present in the analysed system).

- **State support in component models:** The lack of work addressing the discussed issues can be explained by insufficient support of stateful information in existing performance-prediction models. Industrial models (like EJB, CCM or Corba) have been designed to support internal state, since it is one of the crucial implementation details, but lack the support of broad analysis capabilities with respect to system properties. Academic research-oriented stateful component models (like SOFA [7]) are often accompanied with a special analysis method for a set of functional system properties (model checking), but not for performance, which is of our interest. The performance-driven research-oriented component models (see detailed survey in section 3.1) either lack support for state modelling or model state only partially (see table 2).

1.3 The Contribution of the Paper

This paper addresses the challenges via three main contributions: (i) identification of stateful information in component-based systems and their classification into a set of categories, (ii) critical evaluation of state modelling in current performance prediction models, and an extension to a chosen performance-prediction language to provide sufficient state-modelling capability, and (iii) state-dependency analysis discussing performance impact of state-modelling. As a proof of the concept, we provide a realistic case study demonstrating the influences of internal state on a real system measurements compared to prediction based on state modelling abstractions.

The paper is organized as follows. Section 2 realizes the first contribution. It identifies and discusses state-specific properties of component-based systems, localizes stateful information, and classifies them along two dimensions into a categorization. Section 3 realizes the second contribution. It surveys existing performance-driven component-based models with respect to state support, and extends one of the models, Palladio Component Model (PCM), to sufficiently support the identified state categories. Section 4 realizes the third contribution via discussing the observations from a set of experiments performed on stateful PCM models for individual state categories, and section 5 presents results from the analysis of a realistic case study. Finally, section 6 discusses the results and section 7 concludes the paper.

2. STATEFUL COMPONENT-BASED SYSTEMS (SCBSs)

In this paper, we understand the state as an information remembered inside the system. A state is typically context or history-dependent, and is used to navigate system behaviour depending on the current state value. Therefore, a state influences system control flow, which propagates into resource-demand sequences, and finally to performance properties (such as response time, throughput, and resource utilisation). A typical example of a state is an attribute of an object in object-oriented programming, which is used to store information (updated by the methods of the object) and which is employed for customizing object's response to incoming calls.

In literature, two main streams of understanding a state can be found. In the first one [16, 11, 14], the authors attach

a state as an additional information to behavioural models. A state can be used in behavioural decisions. The behavioural model set and read state explicitly. In the second one [7], a state is encoded implicitly in the current position in system execution (behaviour). The main difference between the two is that in the first case, an update of a state is possible, and can be used to adapt the behaviour of the element. In the second case, the state cannot be changed explicitly. When we assume that a system comprises of interacting components, the impact of the state rises in the case of parallel usage of components, when all users share the same stateful information coordinating their behaviour.

2.1 Specifics of CBSs with Respect to a State

The state-relevant information influencing system performance can be found at different stages of system life-cycle. As distinct to classic software systems, the life-cycle of a component-based system constitutes of two separate abstraction lines—life-cycle of a component and life-cycle of a composite system [8, 25]. Moreover, components can be of two types: primitive and composite. Primitive components directly encapsulate implemented functionality, and are typically viewed as black boxes. Composite components are constituted by a composition of existing components, and are often viewed as grey boxes. In a similar fashion, we assume that the state of a composite component is simply a composition (an ordered n-tuple) of the states of its subcomponents. In this sense, a complete composite system has two kinds of states: (i) the implicit state inherited from the (primitive) components in the system, and (ii) an explicit state containing additional information specific to the full system.

Life-cycle stages of a component:

- **Specified component:** represents a component frame with known provided or required interfaces.
- **Implemented component:** defines how the provided services of the implementation call the required services.
- **Instantiated component:** is an identifiable component instance derived from the implemented component, and ready to be executed (in its initial configuration).
- **Deployed component:** is a component instance allocated on a hardware.
- **Running component:** is an actually executed component that serves client requests (not necessarily in its initial configuration).

Life-cycle stages of a composite system

- **Specified system:** is a frame of the system with known access points and services required from the environment.
- **Assembled system:** is an executable system assembled from implemented (instantiated) components, and ready to be launched (in its initial configuration).
- **Deployed system:** is an assembled system deployed on underlying software and hardware.
- **Running system:** is a system at any moment of its execution.

The responsibility to model stateful information and initialise suitable state abstraction is based on CBSE development process. We divide the responsibility to model the state between development roles considering the moment in the development when certain role has enough information to refine the model with required state definition. The overall development process, integrating the evolution on both component and system level can be understood in terms of involved developer roles, which are component developers, software architects, system deployers, and domain experts [18]. *Component developers (CD)* code the components, and annotate their interfaces with abstract behavioural specifications, to facilitate the usage by third parties. *Software architects (SA)* assemble selected components into architectures forming the system. *System deployers (SD)* design the resource environment (e.g. CPUs, network links), and allocate the components in the architecture to the resources. Finally, *domain analysts (DA)* communicate and specify the system-level usage profiles (call frequencies and expected input parameter values), which then can be employed in formal reasoning about system properties.

	Run time	Deployment time	Instantiation time
Component	(a) Protocol state (b) Internal state	(c) Allocation state	(d) Configuration state
System	(e) Global state	(f) Allocation state	(g) Configuration state
User	(h) Session state (i) Persistent state		

Table 1: Identified state categories.

2.2 State Categorisation for CBSs

To find a definition of a state in the context of CBSs and performance predictions, we studied different categories of states in existing component-based systems and component models (see section 3.1). We observed that the notion of component/system state involves various properties and is dependent on different execution processes in the system. With respect to these, we have identified two dimensions, along which we categorise observed state types.

(i) **Place dimension** answers the question: *Is the state proprietary to a component/system/user?*

(ii) **Time dimension** answers the question: *Is the state initialised or changed at run/deployment/instantiation time?*

Table 1 outlines the identified state categories. Along the place dimension, it distinguishes *component-*, *system-* and *user-specific* states, all defined below. With respect to the time dimension, we studied all stages of component-system life-cycle, and observed that a state is by nature a dynamic information that evolves independently for individual elements in the system. If it is fixed along a life-cycle, it is not set before the element gains its identity (instantiation stage in case of a component, assembly stage in case of a system). We refer to this moment as *instantiation time*. The following moments are the *deployment time*, which corresponds to the deployment stage of the life-cycle, and *run time*, which belongs to the run-time stage.

The rest of this section presents the identified state categories, structured to three sections along the place dimension, and for each category, it outlines a demonstrating example, and comments on its modelling.

2.2.1 Component-Specific State

Component-specific state is an information remembered for each component, and used inside the component to adjust component's behaviour to incoming requests. Component state can be modified only by the services of the component, not by other components.

(a) Protocol State: This state holds an information about currently acceptable service calls of a component. It is typically part of an interface contract between service provider and its client [25].

Example: Consider a component managing a file, which can be opened, modified and closed. The component is initially in the state when it accepts only the command for opening the file. After that, it moves to the state, where the file can be either modified or closed. Closing takes the component again to the initial state. The indication for a protocol state performance impact is, for example, rate of rejected requests (contenting the communication link). Analogically, the protocol state uses to be employed also for modelling component life-cycle, including stages like inactive, initialised, replicated, or migrated component.

Modelling: The protocol state uses to be identified by component developer, and attached to a component via a proxy, filtering the calls on component interfaces. Illegal calls are either dropped or returned to the caller with an exception.

(b) Internal State: This state holds an internal information set by the services of the component (at run time), and used to coordinate the behaviour of the component with respect to the current value of the state. Internal state is externally invisible, and externally unchangeable.

Example: Consider a component that can be in either *full* or *compressed* mode, based on the remaining capacity of its database. If it is in the *compressed* mode, all insert queries on the database are additionally compressed.

Modelling: The internal state is defined by component developer, and stored internally as a local variable of each component instance. To reflect the state in a component model, there must be a possibility to define such a local variable, set its value at run time, and query its current value.

(c) Allocation State: This state holds component properties specified at deployment time, based on the allocation environment of the component.

Example: An example of a performance-relevant deployment property is for instance the maximal length of a queue used by the component. Such a property is set at deployment time, and remains fixed along the execution of a component.

Modelling: The component-specific allocation state can be modelled with a static component parameter, and is identified and set by the system deployer role.

(d) Configuration State: This state holds instance-specific component properties, fixed during instantiation of the component.

Example: The configuration state may specify a selected parallel-usage strategy (like rendezvous or barrier synchronization), which may differ for each component instance.

Modelling: Similarly to the component allocation state, the configuration state can be modelled with a static component parameter. In this case, it is typically set by a software architect, who decides on the configuration of the primitive components forming the assembled architecture.

2.2.2 System-Specific State

System-specific state is an information remembered in one copy for the whole system, and used to customize or coordinate joint behaviour of individual components. This state abstraction gains on importance with analysing the state of virtualised systems, cloud computing or systems sharing deployment environment.

(e) Global State: This (run-time) state holds a global information shared and accessed by all components.

Example: A typical example of this kind of state is a global counter, remembering for instance the number of service calls executed in the system since the last back-up of the system, and triggering the back-up process after a certain number is reached.

Modelling: Global state is specified by a software architect during system assembly, in terms of a modifiable system attribute (global variable). It can be either managed directly by the execution environment, or be encapsulated within a component that manages it as its internal state, update its value on request, and answers the questions on its current value.

(f) Allocation State: This state holds deployment-specific information shared by all components in the system.

Example: The examples include the availability of supportive services of the underlying infrastructure (e.g., middleware), parameters of employed thread pool, or selected communication or replication strategies.

Modelling: The system-specific allocation state can be modelled with a static system parameter, and is identified and set by a system deployer.

(g) Configuration State: This state defines system configuration properties specified before launching the system.

Example: This may be for example an upper bound on the number of component instances that may resist in the system at the same time. This is an information of a configuration character, and utilized by all components whenever a new component instance is to be created.

Modelling: The system-specific configuration state can be modelled analogically to the configuration state, and is identified and set by a software architect.

2.2.3 User-Specific State

User-specific state is an information remembered for each user, and used to customize system behaviour to the user.

(h) Session State: This state holds a user-specific information for a single session. The information defining the state is forgotten when the session terminates.

Example: A session can represent one sale performed in a supermarket system. Each sale may start with scanning a customer card, which then customizes system processing of the sale. The system may for instance dynamically recom-

pute during the shopping process the prices of some products or their combination, which may be time consuming and can influence the system response time for a user.

Modelling: This kind of state is derived from the information given by a domain analyst, and could be modelled by additional input parameter in usage model or by more specific component state parameters. The behaviour in system per user/session could depend on the history of actions in the session, this history information could be traced in component internal parameters, what builds together with the persistent state an overlap with component state definition.

(i) Persistent State: This state holds a user-specific information throughout the whole existence of user in the system, independently on an existence of a session belonging to the user.

Example: Each user of an online Media Store has a different limit on data for download under full downloading speed. The system needs to remember this information to control the attempts of users to download data over the limit, and regulate downloading speed accordingly.

Modelling: The persistent state can be modelled analogical to the session state, with a persistent data store involved.

3. PERFORMANCE MODEL FOR SCBSs

This section surveys existing performance-prediction component models with respect to their state-related capabilities, and summarizes their coverage of identified state categories in table 2.

3.1 State of the Art Evaluation

Existing performance-driven component models can be based on their analytical methods classified into four main streams: design-time, formal-specification, measurement, and simulation models.

In the group of *design-time* performance prediction methods these are few that partially support state modelling. First of them is the CB-SPE approach by Bertolino and Mirandola [6] that uses UML extended with SPT annotations profile to model component state or configuration in a static way. The component model based on a proprietary metamodel Palladio Component Model (PCM) [4] builds on statical state abstractions too. Additionally this model allows to model session state through additional input data in an usage profile of a system. Despite of these state abstractions a need of further extensions for state modelling was identified in PCM [17]. The PECT model [14] deals with state modeling in a more detail and addresses the performance predictability properties of components with runtime system assembly variability. Even though the notion of state is partially included there is no full support for including of this state-based variability in performance predictions. This model builds on a Component Composition Language (CCL) that allows to model component behaviour based on statecharts. The performance impact of state is not further investigated, the focus of state modeling is directed on model checking of functional properties. Additionally, based on statecharts and certain behaviour claims, reliability of the system can be verified. Similarly, state is modelled in the Component-Based Modeling Language (CBML) with the possibility to statically configure component parameters. In the component model ProCom [23] designed for embedded systems, state is modelled only statically by a set of component parameters. Further, the component architecture of COMQUAD [20] is using Petri nets as a system behaviour model, however, the dependency of the service call on input data is ommited. A lot of other models claim an ability to express state changes but in many cases they refer to the behavior protocol checking [14], state changes monitoring [21] or performance annotations based on measurements [5].

The *formal specification* model for testing of performance and reliability HAMLET [11] suggest to model state as an additional input (additional floating point external variables loaded in the time of component execution) and provide tests showing functional aspects of a state. The *measurement* approach called AQUA [10] inherently monitors state impact (component description is given by the specification of EJBs) and showed how important it is to understand how system state is interpreted. Another approach to measure EJB applications NICTA [19] provides benchmarking methods to get platform-independent information, such as thread pool size etc. The *simulation-based* approach MIDAS [2] determines performance characteristics of the system through state estimation or computation during simulation, for example queueing characteristics.

3.2 Palladio Component Model (PCM)

Based on the evaluation in the section 3.1 we decided to extend the Palladio Component Model (PCM) [4] with further capabilities to model stateful information. This extension is one of the contributions we introduce in the section 3.2.1. The advantage of this model is its component-based nature, already partial support for state modelling and possibility to model usage profile in detail.

First, the related foundations will be introduced. Figure 2 shows a condensed example of a PCM instance. In this section, we informally describe the features of the PCM meta-model and focus on its capabilities for state modelling. The division of work targeted by CBSE is enforced by the PCM, which structures the modelling task to four independent languages reflecting the responsibilities of the four different developer roles outlined already in section 2.1.

Component developers are responsible for the specification of components, interfaces, and data types. Interfaces are first class entities in the PCM, consist of multiple service signatures, and follow the CORBA IDL syntax. Each provided service of a component is defined by abstract behavioural specification (so-called service effect specification (SEFF)), which abstractly models the usage of required services by the provided service (i.e., external calls), and the consumption of resources during component-internal processing (i.e., internal actions). *Component developers* can annotate external calls as well as control flow constructs with parameter dependencies. This allows the model to be adjusted for different system-level usage profiles. Parameter values can be of different type (e.g., string, int, real, composite) and can be characterised with random values to express the uncertainty when modelling large user groups. Further, each component (or composed component) can have static component parameters defined. *Software architects* compose the component specifications into an architectural model. They create assembly connectors, which connect required interfaces of components to compatible provided interfaces of other components. They usually do not deal with component internals, but instead fully rely on the service effect

Component Model	Design-time prediction models						Formal Specification Methods	Measurement Methods		Simulation Methods
State Category	**CB-SPE**	**PALLADIO**	**PECT**	**CBML**	**PROCOM**	**COMQUAD**	**HAMLET**	**AQUA**	**NICTA**	**MIDAS**
Component-specific										
(a) Protocol	n/a	Prov./req. protocols	Statecharts	CBML	n/a	Petri-nets	n/a	n/a	n/a	n/a
(b) Internal	n/a	n/a	n/a	n/a	n/a	?	?	n/a	n/a	n/a
(c) Allocation	RT-UML PA profile Annotation	Static parameter, Passive Resources	n/a	?	Static parameter	n/a	?	Static parameter	Static parameter	Static parameter
(d) Configuration			n/a	Static parameter		n/a	Additional input			
System-specific										
(e) Global	n/a	n/a	Statecharts	CBML	n/a	n/a	?	n/a	n/a	n/a
(f) Allocation	n/a	Static parameter, Passive Resources	n/a	n/a	n/a	n/a	n/a	?	?	Static parameter
(g) Configuration	n/a		n/a	n/a	n/a	n/a	n/a	?	?	
User-specific										
(h) Session	n/a	Input data	n/a	n/a	n/a	?	Additional input	Static parameter	?	?
(i) Persistent	n/a	n/a	n/a	n/a	n/a	n/a	n/a		?	?

Table 2: Component Performance Models Comparison.

specifications supplied by the component developers. Furthermore, software architects define the system boundaries and expose some of the provided interfaces to be accessible by users. *System deployers* model the resource environment (e.g., CPUs, network links) and allocate the components in the architectural model to the resources. Resources have different attributes, such as processing rates or scheduling policies. Finally, *domain experts* specify the system-level usage model in terms of stochastic call frequencies and input parameter values for each called service, which then can be automatically propagated through the whole model and define non-persistent user session parameters.

The PCM already provides certain abstractions or approximations to model state: (i) static component parameters (or properties) characterize the state of a component in an abstract and static way and hence offer a more flexible parameterization of the model. These parameters are propagated through development process differently, they are defined and initialized by a *component developer* and can not be changed at runtime. (ii) Limited passive resources, such as semaphores, threads from a pool, or memory buffers result in waiting delays and contentions due to concurrently executed services. (iii) Input data from usage profile allows to express session state. Table 2 illustrates the capabilities of PCM to model identified state categories.

3.2.1 PCM Stateful Extension

We extended the component behaviour model of the PCM (the SEFF) to allow the modelling of component internal state as described in section 2.2.1(b). With this extension, also system specific global state (cf. section 2.2.2(e)) can be modelled by adding a blackboard component that makes its internal state available to other components in the system. Only two additions to the PCM metamodel are required to model component internal state and global system state. First, we declare a set of state variables for a component. Only a declared state variables can be used within a SEFF. Second, we add a `SetStateAction` to the SEFF, which allows to set the state variable to a given expression. Input data of the SEFF, other state variable values and the previous state variable value can be used in the expression. Now, the state variable can be used in branch conditions or resource demands as a parameter. The use of *PCM Stateful* extension is illustrated in section 5.

We can analyse an extended PCM model with an extended version of the *SimuCom* simulation presented in [4] to obtain the performance metrics. At simulation runtime, each component is instantiated and holds its state variables. When a `SetStateAction` is evaluated, its expression is evaluated and stored in the state variable. If `BranchActions` and `InternalActions` access state variables, the value is retrieved. The extension increases the expression power of SEFFs and allows programming, although the language does not become Turing complete (all loops are bounded). As multiple requests to the system are analysed concurrently, we can encounter race conditions and resulting unexpected behaviour. In our example above, race conditions are excluded because the branch condition and `SetStateAction` are evaluated in the same simulation event (no time passes in simulation). However, in general, if a resource demand is executed between reading the state in a `BranchAction` and setting the state in one of the branches, both actions are executed in separate simulation events. Here, a second request to the component could read or change the state in between, leading to race conditions. With the extended state modelling, steady-state behaviour is not guaranteed any more. While this limits analysability, it also can help to detect problems in a software design.

For example, assume a system service that becomes the more expensive the more requests have been served. Then, the response time of the system will ever increase ("The Ramp" antipattern [24]) and no steady state can be reached.

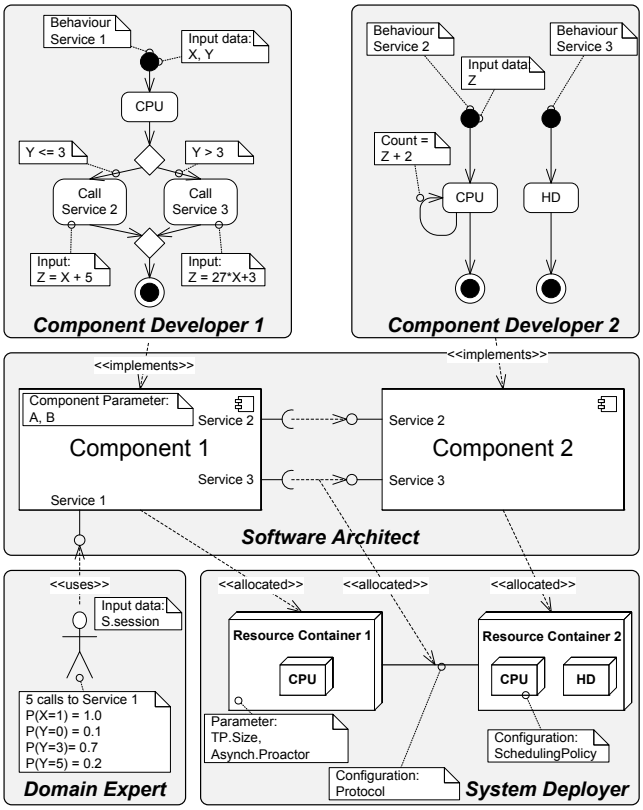

Figure 2: PCM example.

With the extended state modelling, this performance antipatterns can be detected in the simulation results.

4. STATE DEPENDENCY ANALYSIS

After identifying state types in component-based systems, and extending the PCM performance-prediction model to support them, this section elaborates the third contribution of the paper—a study of the performance impact of identified state types, and observations about the influences that should drive the decision on the abstraction level of state modelling. The states in the system should be modelled only if the increase in the accuracy they bring outbalances the price that needs to be paid for the increased model complexity [17].

In design-time performance prediction, this issue has already been addressed for various other constructs, including service parameters, return values, or usage-profile propagation. This section gives an insight into the issue for state modelling, which has not been addressed so far, and hence tries to help the software engineers to find the balance between accuracy and complexity of models more competently.

4.1 Quantification of Model Complexity

The complexity of a model can be best understood when translated to a low-level formal language with clearly definable size. One of the formalisms most commonly employed for this purpose are labelled transition systems. In the case of component-based systems are particularly different kinds of interacting automata [28, 27] used. Allowing to form labelled transition systems via composition of automata-based models of individual components. In [27], the inclusion of a stateful information in a model is studied in terms of *Component-Interaction Automata*. It is shown that a component/system state can be encoded as an automaton interacting with the automata for component services—answering their queries of its current value, and accepting their commands to change the value. The model of a system is then a composition of not only the models of individual services (implemented by the components), but includes also the models of all states (whose size correspond to the number of possible state values).

Since the size of a composite component-interaction automaton is defined over a cartesian product of the vertices of composed automata, the size of the composite model can be in the worst case a multiplication of the initial stateless model with the size of the internal-state model. We have observed, however, that not only that this case is very unlikely to occur, but the model that includes stateful information can be even smaller than the initial stateless model, due to higher certainty about future behaviour of the system.

4.2 Diversity Among State Categories

In section 2.2 we have identified nine state categories. Though they all embody the same construct (defined in section 2) and theoretically bring analogical performance impact, their practically observed performance impact differs, and is influenced by diverse criteria. However, one can also observe strong similarities among some of the categories:

- **Allocation vs. Configuration state:** Both the allocation and configuration state (consider the component-specific case for now) are fixed before the actual system execution. Thus from the performance point of view, both of them can be understood as fixed component parameters, often usable in an interchangeable way.

- **System vs. Component-specific states:** Even if the component-based system behaviour is encapsulated in components, and structured to architectures, its core is in the interaction of system services. If we abstract from component boundaries, we can find a strong analogy between component internal state and system global state, and between component- and system-specific allocation and configuration state.

- **Session vs. Persistent state:** Note that from the point of view of performance analysis, the persistent state can be understood as a session state for one life-lasting session.

The identified similarities separate the defined state types into four classes, with the following representatives[1]: (a) Protocol state, (b) Internal state, (c) Allocation state, and (h) Session state.

4.3 Performance Impact of State Classes

For each of the four representatives identified above, we performed a number of experiments and draw a number of observations on the performance impact and costs of the state class, which we present in this section. For both the performance impact and the cost, we compared the stateful

[1]Any other members of the classes could be chosen as the representative, we chose the first ones from the line of their definition.

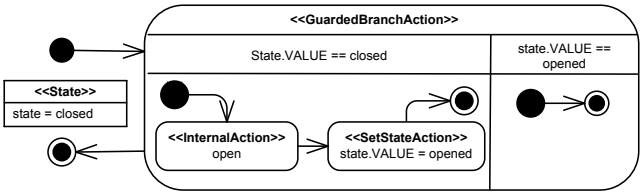

Figure 3: A SEFF of `open()`.

model of a PCM instance to the stateless model of the same example, where the state-dependent decisions are guarded by probabilities.

(a) Protocol State: Recall the protocol-state example outlined in section 2.2. The protocol state in the example can have two values: *closed*, when the only acceptable call is `open()`, and *opened*, when the component can accept calls `modify()` and `close()`. Both the stateful and probabilistic model of the example in PCM consist of three SEFF models and one usage profile. Each SEFF starts with a branch condition deciding if the service is going to be executed or rejected (see figure 3). While in the stateful version, the branch is guarded by a current value of the protocol state, updated after executing `open()` and `close()`, the probabilistic model fixes the probabilities of the branches based on expected likehood of the alternatives.

Performance impact: A number of performed experiments with different variations of the probabilistic model showed two important observations about accuracy of the stateful model comparing to the stateless model:

- The performance impact of the protocol-state modelling highly depends on the a-priori knowledge of the usage profile, which in general cannot be guaranteed since component behaviour and usage profile are defined independently by different developer roles.
- Even if the usage profile is known, the actual probabilities of service triggering depend on component's environment through which the usage profile is propagated, and thus can be very hard to quantify.

We can conclude that the importance of protocol-state modelling raises with the lower knowledge of usage profile, and higher complexity of component's environment.

Model-size costs: The stateful model of each service has a unified form, having two independent alternatives: the first (complex one) if the service is executable, and the second (trivial one) if the call is rejected (see figure 3). In a stateful model of such a service, two sources of model-size increase can be identified.

- An increase due to state update after service execution, which is negligible.
- An increase due to remembering the actual state value, and accordingly executing only the right alternative. If the size of the model is understood in terms of a labelled transition system (a graph describing the paths of possible system behaviour), then the size remains unchanged as far as there is always only one state value for which each service can be executed. If a service can be executed in more than one values of the protocol state, the number of vertices in the model can be multiplied with the number of such state values. On the other hand, the complexity of the paths throughout the transition system remains unchanged.

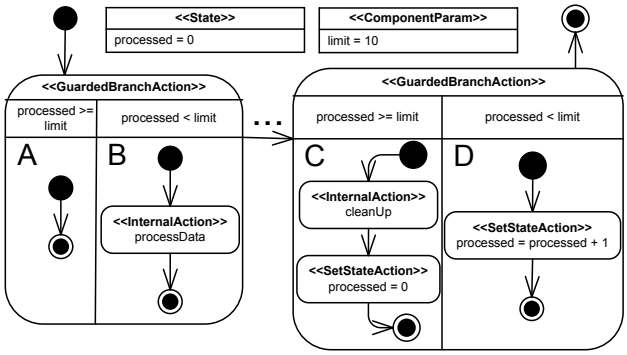

Figure 4: A SEFF of `processData()`.

(b) Internal State: Consider an example outlined in figure 4, with internal state *processed* remembering the amount of processed data, and coordinating a component to either process additional data or perform cleanup. A probabilistic model would be analogical, with the branches guarded with the probabilities of state values.

Performance impact: This example was selected to disclose an additional influencing factor (besides the two identified for the protocol state), specific to this state category. It is connected to a possible correlation of state values in subsequent branches guarded by the internal state (typically with additional execution in between of the branches). Recall the example in figure 4 with strongly positively correlated branches (let us denote the alternatives in the first branch A and B, and in the second branch C and D). Note that while in the stateful model, there are only two possible service executions (either A followed by C, or B followed by D), in the probabilistic model, four alternatives are possible (both A and B can be followed by anything). The observation can be summarized as follows:

- The importance of internal-state modelling raises with the higher correlation of subsequent state-driven decisions. In such a situation, it is very unlikely that the system can be modelled faithfully without the internal-state construct.

Model-size costs: The model of a service involving internal state can have much more variability than in the case of protocol state, since the state-guarded branches and state updates can be present anywhere in the model. This in the worst case implies multiplication of the model size with the size of the state (number of its possible values). In practice however, this case is very unlikely to occur. The likehood is decreased by the following factors:

- A high connection of component behaviour to a state value. The model does not grow to the worst case if some of the behaviours are possible only under a particular state value. Then the combinations of these behaviours with the infeasible state values do not appear in the model and restrict the size increase (analogically to the argument for the protocol state).
- A low number of independent state-guarded branches. On the other hand, a high number of independent branches does not increase the number of vertices in the model, but increases the number of transitions,

and hence the number and complexity of behaviour-describing paths.
- A small number of state updates.

(c) Allocation State: Recall the allocation-state examples outlined in 2.2. Their most important property (shared with the configuration state) is that they are fixed at deployment time (resp. instantiation time) and do not change during system execution.

Performance impact: Thanks to this specific, there are two main observations influencing the effect of allocation-state modelling:

- As distinct to so far discussed categories, the general influence of the allocation state to system performance is independent on the usage and the environment. This is because the state-guarded branches are evaluated in a fixed way, irrespective of the usage.
- On the other hand, the prediction accuracy is critically dependent on the knowledge of component/system deployment parameters, which allows to cut off the behavioural branches that go against the value of the parameter. When such an information is not available to the component developer (since it is determined by a different role), the probabilistic model exhibits high inaccuracies.

Model-size costs: As the value of the allocation state does not change along system execution, there is no increase in model size, quite the contrary. Since all infeasible branches are never executed, the reachable space of the stateful model is even smaller than in its probabilistic variant.

(h) Session State: Consider the session-state example from section 2.2, with sessions connected to individual sales, parameterized by an information about the customer. The PCM model can be very simple, propagating the user-specific state in terms of an input value throughout the whole session.

Performance impact: The session state exhibits some similarities, but also differences to all the state classes discussed above. It is very similar to the allocation state, but is not fixed along the whole execution (differs for individual sessions). It changes very rarely, and is updated only on a specific place (similarly to the protocol state). On the other hand, it may guard behavioural branches anywhere in the execution, as distinct to the protocol state but similar to the internal state. This implies the following:

- The impact is not very dependent on the usage profile and environment, but highly dependent on the knowledge of the distribution of the session state values (similarly to the knowledge of deployment parameters in case of the allocation state).
- Since the subsequent queries on the state value are highly correlated, probabilistic models can hardly model session-state dependent behaviour faithfully.

In summary, this construct is crucial in the model, because of strong correlation of subsequent state-guarded branches, and changeability of the state value along the system execution.

Model-size costs: The observations on the size of the model follow:

- The increase due to remembering the actual state value is similarly to the internal state dependent on the connection of component behaviour to the state value. The weaker the connection is, the closer the model can grow to the worst case.
- Thanks to the correlation of subsequent branches, there is basically no complexity increase in terms of the behavioural paths.
- There is basically no size increase due to state update, since the state is not updated inside the system, and occurs very rarely.

5. VALIDATION

In this section, we use the state model introduced in section 3.2.1 to predict the delivery time of transactional messages observed on our motivating example in section 1.1. Transactional messages are common in today's enterprise applications, such as implemented by *SPECjms2007 Benchmark* [9]. However, the transactions used in the supply chain management supermarket of the benchmark are limited to small, predefined transaction sizes. To provide a better evaluation, we implemented an application that allows to configure the number of messages send in one transaction following the philosophy of SPECjms2007. We excluded external disturbances (such as database accesses) and focussed on the evaluation of the messaging system. For performance prediction, we extended our performance completion for message-oriented middleware called messaging completion in the following [13]. The messaging completion subsumes several components that reflect the influence of different middleware configurations such as guaranteed delivery, competing consumers, or selective consumers. A model to model transformation (implemented in QVT Relational [22]) generates the necessary completion components and integrates them into the software architecture. We have already demonstrated that the messaging completion can predict the performance of a SPECjms2007 scenario with an accuracy of 5% to 10% [13]. In the subsequent paragraphs, we present an extension of our messaging completion that enables the prediction of influences of transactions on the delivery time of a message.

Completion for Message-oriented Middleware: Figure 5 shows the components and connections that are generated by the messaging completion (see [13] for details). The completion consists of *adapter components* and *middleware components*. The first forwards requests and calls the middleware components that issue platform-specific resource demands. The `Marshalling` component computes the message size based on the method's signature. The message size is passed to subsequent adapters as an additional parameter, so that the original interface (`IFoo`) needs to be extended (`IFoo'`). The `Sender Adapter` calls the `Sender Middleware` which loads the resources of the sender's node and forks the call to the `MOM Adapter` to reflect the asynchronous behaviour of the messaging system. The `MOM Adapter` realises the transactional behaviour of the messaging system. The `Receiver Adapter` calls the `Receiver Middleware` and, thus, loads the resources of the receiver's node. It forwards the requests to `Demarshalling` which maps the extended interface (`IFoo'`) back to the original interface (`IFoo`).

Modelling transactional behaviour of the `MOM Adapter`: In order to start a transaction, the sender has to explicitly call method `startTransaction`. Its behaviour (see fig-

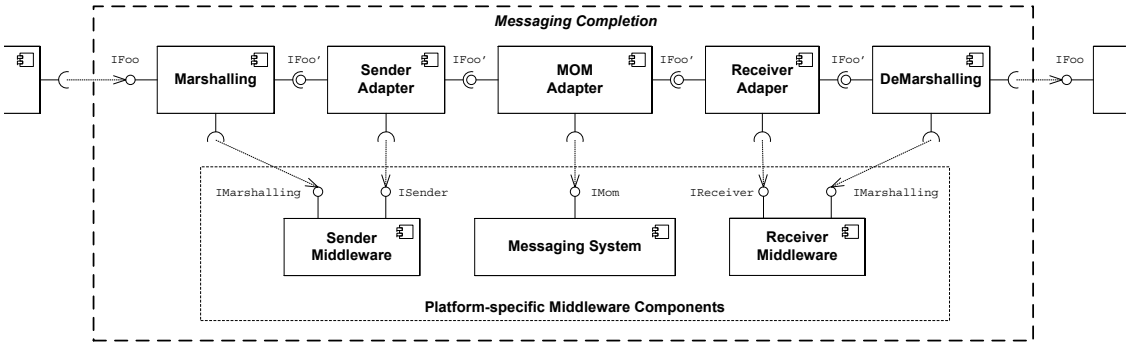

Figure 5: Components of the MOM completion.

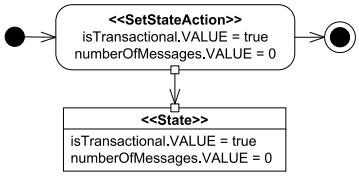

Figure 6: Starting a new transaction.

ure 6) consists of a single `SetStateAction`, which resets the number of messages to zero (`numberOfMessages.VALUE = 0`) and enables the transactional message transfer (`isTransactional.VALUE = true`). When `startTransaction` has been called, all messages send in the following will be part of the transaction until `commitTransaction` is executed. The behaviour of the `MOM Adapter` varies for transactional and non-transactional messages (see figure 7). If the message is not part of a transaction, the adapter simply calls the `Messaging System`, which loads its local resources with the service demands necessary for transferring the message, and forwards the messages. Otherwise, if the message is part of a transaction, then the `MOM Adapter` increases the current number of messages of the transaction (numberOfMessages.VALUE = numberOfMessage.VALUE + 1) and queues the message. The queueing is modelled by two actions. The first external call action (`IMOM.queueMessage`) loads the resources of the `Messaging System`. The second action acquires the passive resource `transactionQueue`, which blocks the message transfer until the `transactionQueue` is released. When the transaction is committed and the messages blocked at the `transactionQueue` are released, the `MOM Adapter` processes the message transfer (`IMOM.processMessageTransfer`). Furthermore, it notifies the behaviour of `commitTransaction` that the message has been transfered (`transferCompleted` is released). Finally, the `MOM Adapter` forwards the message to the `Receiver Adapter`. This behaviour ensures that all messages are delivered in the same order as they have been send. Figure 8 shows the behaviour executed to commit a transaction. The RD-SEFF reflects the successful execution of a transaction and neglects possible rollbacks and re-executions. To commit a transaction and deliver all messages to the receivers, a loop action iterates over all messages blocked during the transaction (`numberOfMessages.VALUE`). For each message, it unblocks its transfer (releases passive resource `transactionQueue`. To ensure the sequential delivery of messages, it waits for the successful transfer of the message (aquires passive resource

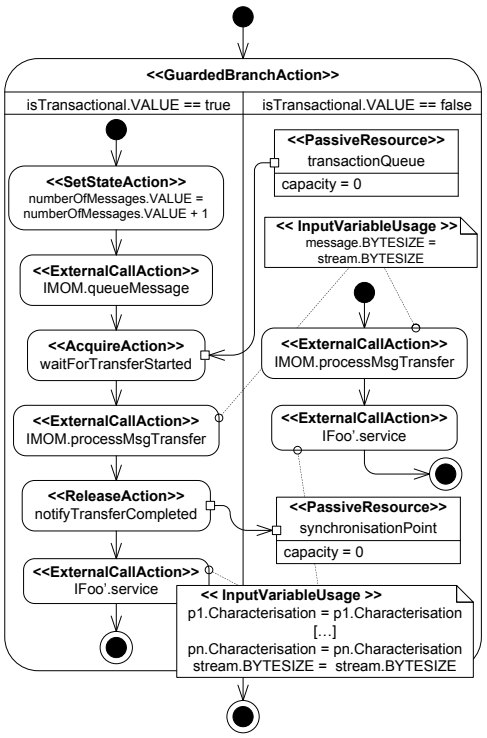

Figure 7: MOM Adapter: Message Transfer.

`synchronisationPoint`) before it continues. Finally, the transaction is terminated (`isTransactional.VALUE = false`) and the number of queued messages is reset (`numberOfMessages.VALUE = 0`).

Results: Figure 9 shows the prediction results for transactional messages using the models presented in this section. The corresponding real measurement is shown in figure 1. The predictions correctly reflect the dependency of a message's delivery time on its position in the transaction. Furthermore, the predicted delivery times range from 400 ms to 1400 ms which corresponds to the observed delivery times. Table 3 lists the predicted and measured median values for different transaction sizes. Due to the high variance of the delivery times, the median serves as a representative value for a specific transaction size. However, the median can only be considered as an indicator for the prediction accuracy. In table 3, predictions and measurements deviate less than 4%.

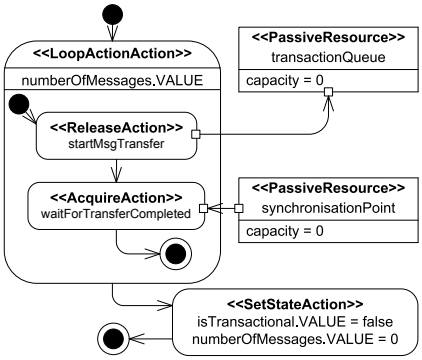

Figure 8: MOM Adapter: Commit Transaction.

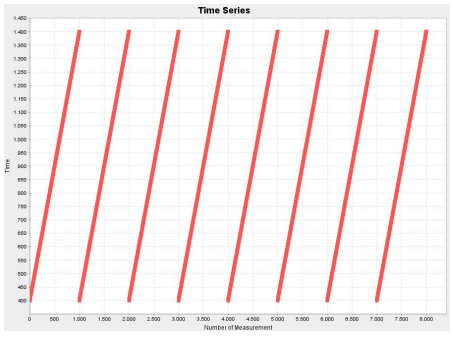

Figure 9: Predicted delivery times for messages.

These results indicate, that the extension of our messaging completion based on *PCM Stateful* can accurately predict the influence of (successfully completed) transactions on the delivery time of a message.

6. DISCUSSION

The increased expressiveness of stateful models comes at a cost. Stateful models may have much higher complexity and size, which may complicate their analysis. Even if the models are not analysed fully, and are examined with simulation methods (like in the case of PCM), model complexity may have an impact on the time needed for sufficiently accurate performance prediction (duration of a simulation run). The time necessary to execute a simulation run is further influenced by the variability of simulation results. The state-dependent system variability mirrors in the variance of the results and consequently influences the number of measurements necessary to achieve results with a high confidence. The cost of a single simulation measurement depends on the length of the simulated trace. Explicitly modelled states have only little effect on the length of simulation traces, which mainly depend on the modelled software architecture (e.g., loops dependent on a state value). On the other hand, one should to keep in mind that the confidence about the correctness of predicted values will be higher if a low-coverage simulation is run on a more accurate (stateful) model, than if a high-coverage simulation is run on an unrealistic (stateless) model.

Transaction Size	Measurement (Median)	Prediction (Median)
1	1,665 917ms	-
2	2,506 566ms	2,609 999ms
4	4,157 104ms	4,619 999ms
10	9,145 595ms	9,050 000ms
20	17,012 373ms	17,079 999ms
100	82,752 583ms	85,440 000ms
400	356,843 626ms	360,980 000ms
1000	943,539 863ms	943,370 000ms

Table 3: Measurement/Prediction Comparison.

When studying the performance impact of state modelling in section 4, we have compared stateful models to their approximations with probabilistic models. It is clear that if the influence of state is known a priori, probabilistic models can approximate the stateful models very closely. In order to achieve this, all states and their influence on performance must be known in advance. For example, the influence of transactions (described in section 5) can be approximated probabilistically, if the waiting time of a message is known and modelled as an explicit delay that depends on the number of messages sent within the transaction. To achieve this, performance analysts have to know in advance the number of messages in a transaction as well as the influence of a message on the transaction's delay. Modelling transactional messages probabilistically results in a comparable distribution of response times. However, the model does not reflect the stochastic dependency of sequentially arriving messages. Furthermore, it provides less flexibility since delays caused by transactional behaviour have to be known in advance. In most cases, such information is not available or the delays are changing constantly. In these cases, an explicit state model eases the design of performance models and allows accurate predictions with the necessary flexibility. Additionally, approximating state by a probabilistic abstraction results in decreased possibility of reuse of the component prediction model because the probabilities are specific for one system, one allocation and one usage profile.

7. CONCLUSION

The paper addresses the challenges of performance prediction for stateful component-based software systems. To achieve this aim, we have elaborated a number of tasks. We investigated the requirements and the offered expressiveness of prediction models for stateful systems. We surveyed the state of the art and extracted a classification scheme of various state-defining and state-dependent model parameters. After that, we critically evaluated the possibility of modelling introduced categories using state abstractions in current performance prediction models. As a result, we extended *Palladio Component Model* to provide sufficient state-modelling capability, and evaluated the benefits and costs it brings in a state-dependency analysis. In the state-dependency analysis, we further identified the similarities and differences of individual state categories with respect to their performance impact and model-size increase, and presented our observations gained from a number of performed experiments.

The future work includes further analysis of user-specific persistent session state, its abstractions and propagating this state dependency on a usage profile through the hierarchy of the model. Similarly, a deeper analysis how different state

types affect the prediction accuracy of introduced modelling methods is needed. The next question is how to define state of the system assembly (e.g., service composition) automatically. The analysis of composability of the state abstractions on the component level to the state abstraction on the system level will answer this question. New challenges rise from the introduction of dynamic architectures and support of virtualisation scenarios and dynamic allocation. Additionally, a set of more detailed analyses based on industrial case studies illustrating impact of state introduction could disclose new interesting observations.

Acknowledgment

The work has been partially supported by the Academy of Sciences of the Czech Republic (grant No. 1ET400300504).

8. REFERENCES

[1] Java System Message Queue -Version 4.3. http://www.sun.com/software/products/message-queue/index.xml, last retrieved: July 2009.

[2] R. Bagrodia and C. Shen. Midas: Integrated design and simulation of distributed systems. *Transactions on Software Engineering*, 1991.

[3] S. Becker, J. Happe, and H. Koziolek. Putting Components into Context: Supporting QoS-Predictions with an explicit Context Model. In *Proc. of Workshop on Component Oriented Programming (WCOP)*, 2006.

[4] S. Becker, H. Koziolek, and R. Reussner. The palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 2009.

[5] A. Bertolino and R. Mirandola. Modeling and analysis of non-functional properties in component-based systems. Elsevier, 2003.

[6] A. Bertolino and R. Mirandola. *CB-SPE Tool: Putting Component-Based Performance Engineering into Practice*. LNCS. Springer, 2004.

[7] T. Bures, P. Hnetynka, and F. Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *Proc. of Conference on Software Engineering Research, Management and Applications (SERA)*. IEEE, 2006.

[8] J. Cheesman and J. Daniels. *UML Components: A Simple Process for Specifying Component-based Software*. 2000.

[9] S. P. E. Corp. SPECjms2007 Benchmark. http://www.spec.org/jms2007/, last visit: January, 2009, 2007.

[10] A. Diaconescu and J. Murphy. Automating the performance management of component-based enterprise systems through the use of redundancy. In *Proc. of Conference on Automated software engineering (ASE)*. IEEE, 2005.

[11] D. Hamlet. Subdomain testing of units and systems with state. In *Proc. of symposium on Software testing and analysis (ISSTA)*. ACM, 2006.

[12] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout. Java Message Service Specification -Version 1.1. http://java.sun.com/products/jms/, last retrieved: January 2009.

[13] J. Happe, S. Becker, C. Rathfelder, H. Friedrich, and R. H. Reussner. Parametric Performance Completions for Model-Driven Performance Prediction. *Performance Evaluation*, 2009.

[14] S. Hissam, G. Moreno, J. Stafford, and K. Wallnau. Enabling predictable assembly. *Journal of Systems and Software*, 2003.

[15] H. Koziolek. *Parameter Dependencies for Reusable Performance Specifications of Software Components*. PhD thesis, University of Oldenburg, 2008.

[16] H. Koziolek. Performance evaluation for component-based software systems: A survey. *Submitted to Performance Evaluation (Special Issue on Software and Performance)*, 2009.

[17] H. Koziolek and S. Becker. Transforming Operational Profiles of Software Components for Quality of Service Predictions. In *Proc. of Workshop on Component Oriented Programming (WCOP)*, 2005.

[18] H. Koziolek and J. Happe. A QoS Driven Development Process Model for Component-Based Software Systems. In *Proc. of Symposium on Component-Based Software Engineering (CBSE)*. Springer, 2006.

[19] Y. Liu, A. Fekete, and I. Gorton. Design-level performance prediction of component-based applications. *Transactions on Software Engineering*, 2005.

[20] M. Meyerhöfer and K. Meyer-Wegener. Estimating non-functional properties of component-based software based on resource consumption. *Electronic Notes in Theoretical Computer Science*, 2005.

[21] A. Mos and J. Murphy. Performance management in component-oriented systems using a model driven architecture approach. In *Proc. of Enterprise Distributed Object Computing Conference*. IEEE, 2002.

[22] Object Management Group. *MOF 2.0 Query/View/Transformation, version 1.0*, 2008.

[23] S. Sentilles, A. Vulgarakis, T. Bureš, J. Carlson, and I. Crnković. A component model for control-intensive distributed embedded systems. In *11th International Symposium on Component-Based Software Engineering (CBSE)*. Springer, 2008.

[24] C. U. Smith and L. G. Williams. New software performance antipatterns: More ways to shoot yourself in the foot. In *Proc. of International CMG Conference*. Computer Measurement Group, 2002.

[25] C. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 2002.

[26] M. Woodside, G. Franks, and D. C. Petriu. The Future of Software Performance Engineering. In *Proc. of Conference on Software Engineering (ICSE)*. IEEE, 2007.

[27] B. Zimmerova. *Modelling and Formal Analysis of Component-Based Systems in View of Component Interaction*. PhD thesis, Masaryk University, Czech Republic, 2008.

[28] B. Zimmerova et al. *The Common Component Modeling Example: Comparing Software Component Models*, chapter 7. LNCS. Springer, 2008.