Softw Syst Model (2014) 13:1319–1343
DOI 10.1007/s10270-013-0336-6

SPECIAL SECTION PAPER

# Stateful component-based performance models

**Lucia Happe · Barbora Buhnova · Ralf Reussner**

Received: 14 February 2012 / Revised: 1 February 2013 / Accepted: 4 March 2013 / Published online: 5 April 2013


**Abstract** The accuracy of performance-prediction models is crucial for widespread adoption of performance prediction in industry. One of the essential accuracy-influencing aspects of software systems is the dependence of system behaviour on a configuration, context or history related state of the system, typically reflected with a (persistent) system attribute. Even in the domain of component-based software engineering, the presence of state-reflecting attributes (the so-called internal states) is a natural ingredient of the systems, implying the existence of stateful services, stateful components and stateful systems as such. Currently, there is no consensus on the definition or method to include state-related information in component-based prediction models. Besides the task to identify and localise different types of stateful information across component-based software architecture, the issue is to balance the expressiveness and complexity of prediction models via an effective abstraction of state modelling. In this paper, we identify and classify stateful information in component-based software systems, study the performance impact of the individual state categories, and discuss the costs of their modelling in terms of the increased model size. The observations are formulated into a set of heuristics-guiding software engineers in state modelling. Finally, practical effect of state modelling on software performance is evaluated on a real-world case study, the SPECjms2007 Benchmark. The observed deviation of measurements and predictions was significantly decreased by more precise models of stateful dependencies.

**Keywords** Stateful components · Performance prediction · Prediction accuracy

Communicated by Prof. Dr. Dorina Petriu and Dr. Jens Happe.

L. Happe (✉) · R. Reussner
Karlsruhe Institute of Technology, Karlsruhe, Germany
e-mail: kapova@kit.eud; kapova@iru.uka.de

R. Reussner
e-mail: reussner@kit.edu

B. Buhnova
Masaryk University, Brno, Czech Republic
e-mail: buhnova@fi.muni.cz

## 1 Introduction

During the last few years, many approaches dealing with performance prediction have been introduced [2]. In the area of *Component-Based Software Engineering (CBSE)*, specialised prediction approaches aim to understand the performance (i.e. response time, throughput, resource utilisation) of the component-based architecture based on performance properties of individual components [3,19]. The performance properties of a component-based system are influenced by a number of factors, including components implementation, deployment, environmental context and system usage. The difficulty of understanding component-based system performance is further strengthened by the influence of the internal state of the system or its components [19]. Such a state can be for instance determined by an internal component attribute indicating if the component should at the moment prioritise reliability (apply a more reliable but slower algorithm) or performance (apply less reliable but faster algorithm).

### 1.1 Challenges of stateful analysis

In the context of component-based performance models, the primary challenge of stateful analysis is the state identification across the component-based architecture and the

decision on an effective state abstraction in the models. In this respect, the following three issues can be identified.

State definition: The property of statefulness can be identified in various artefacts along the component-based system life cycle and in various elements of a component-based architecture. Existing literature lacks the localisation of state-holding information in component-based systems [4,19,36], and its classification into a set of categories.

Performance impact: The benefits of state modelling include increased expressive power of the models and higher accuracy of predictions. The increase of prediction accuracy achieved by state modelling, especially in comparison to the increased effort of modelling and analysis, is however not well studied, as observed by a number of authors [5,19,36]. A discussion on how the existing performance-driven models deal with the interpretation and analysis of stateful-prediction models is elaborated in Sect. 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 the state-related information present in the analysed system shall be included in the model).

The lack of work addressing the discussed issues can be explained by insufficient support of state-related information in existing performance-prediction models. Industrial models (like EJB [33], COM [25] or Corba [27]) have been designed to support internal state of components, since it is one of the crucial implementation details, but lack the support of broad analysis capabilities with respect to system performance. The performance-driven research-oriented component models (see detailed survey in Sect. 3.1) either lack support of state modelling or model state-related attributes only partially (see Table 2).

### 1.2 The contribution of the paper

Besides the identification of state-related information in component-based systems and its classification into a set of categories, the main contribution of this paper is a novel state-effect analysis evaluating the performance impact of the identified state classes together with the discussion of the increase in the prediction difficulty introduced by state modelling. As a proof of the concept, we extended an existing performance prediction framework and employed it in a real-world case study, demonstrating the effect of state modelling on software performance. The paper is an extension of our previous work [16] with an expanded state-effect analysis, an extensive experimental evaluation of newly derived heuristics, and an additional case study.

The paper is organised as follows. Section 2 localises state-related information in component-based systems and classifies it along two dimensions into a set of categories. Section 3 surveys existing performance-driven component models with respect to state support, and extends the selected Palladio Component Model (PCM) [5] to support the identified state categories. Sections 4 and 5 elaborate the main contribution of the paper by introducing an approach supporting software engineers with the information about performance impact and model-size costs of the individual state categories. Section 6 presents a validation of our approach on a realistic case study and Sect. 7 summarises our practical experience and observations from the conducted experiments. Finally, Sect. 8 concludes the paper.

## 2 State categories in component-based systems

In this paper, we use the term *state* to refer to a configuration, context or history related information remembered inside the system (typically as a value of a component/system attribute) and used to navigate system behaviour. Therefore, a state influences system control flow, which propagates into resource-demand sequences, and finally to performance properties (such as response time, throughput, resource utilisation). A typical example of a *state* is the value of an attribute stored inside an object in object-oriented programming, which is accessed by object's methods and used to customise the object's response to incoming calls.

The state as understood in this paper should not be confused with an implicit state of system execution, i.e. the current position in system execution, as used by many approaches employing state-based models (such as Markovian models [10] or finite automata [9]). In our approach, the state is a value of an explicit attribute attached to the behavioural model of system element [11,15,19]. The state can be set and read explicitly, and used in behavioural decisions.

To identify the relevant state-related information across the component-based system architecture, we surveyed existing component-based systems and component models (see Sect. 3.1) and observed that the notion of component-based system state is dependent on various architectural elements and execution processes in the system. In particular, we identify two dimensions, along which we categorise the observed state types.

(i) Scope 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** Identified state categories

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

Table 1 outlines the identified state categories. Along the scope dimension, it distinguishes *component-*, *system-* and *user-specific* states, all defined below. With respect to the time dimension, we examined all the stages of component-based system life cycle [16], 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 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* and *runtime*, which correspond to the deployment stage and runtime stage of the life cycle.

The rest of this section presents the identified state categories, structured into three sections along the scope dimension, and for each category, it outlines a demonstration example.

### 2.1 Component-specific state

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

(a) Protocol state: This state holds the information about currently acceptable service calls of a component. It is typically part of an interface contract between service provider and its client [34].

*Example* Consider a software component managing a file, which can be opened, modified and closed. The component is initially in the state when it accepts only the request 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. An indication for the protocol-state performance impact in this example is, for example, the rate of rejected requests (occupying the communication link). Moreover, the protocol state is often used to reflect component life-cycle stages, such as inactive, initialised, replicated, or migrated.

(b) Internal state: This state holds an internal information set by the services of the component (at runtime) 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 software 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.

(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 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.

(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 synchronisation), which may differ for each component instance.

### 2.2 System-specific state

*System-specific state* is a shared information available to the whole system and used to customise or coordinate joint behaviour of individual components.

(e) Global state: This (runtime) state holds a global information shared and accessed by all components of the system.

*Example* A typical example is a global counter, remembering 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.

(f) Allocation state: This state holds a 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.

(g) Configuration state: This state defines a system configuration property specified before launching the system.

*Example* An upper bound on the number of component instances that may exist in the system at the same time. Such a pre-configured information may be utilised by all components whenever a new component instance is to be created.

### 2.3 User-specific state

*User-specific state* is an information remembered for each user and used to customise system behaviour for 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 customises system processing of the sale. The system may dynamically recompute 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.

(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 may have 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.

## 3 Performance model for SCBSs

This section surveys and compares existing component-based models for performance prediction with respect to their capabilities related to state and summarises their coverage of identified state categories in Table 2. Additionally, we summarise the means the models provide to express the state-related information.

### 3.1 State of the art evaluation

As summarised in Table 2, 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. Each of these approaches utilises different model elements to express state-related information.

Among *design time* performance prediction methods, the following methods partially support state modelling. First, the CB-SPE approach by Bertolino and Mirandola [7] uses UML extended with SPT annotations profile to model a component state or configuration in a static way. The component model, based on a proprietary metamodel PCM (PCM) [5], builds on static abstraction of state modelling, too. Additionally, this model allows modelling a session state through additional input data in a usage profile of a system. Despite these abstractions, a need of further extensions for state modelling was identified in PCM [20].

The PECT model [15] deals with state modelling in more detail and addresses the performance predictability properties of components with runtime system assembly variability. Even though the notion of a state is partially included (see Table 2), there is no full support for simulation of this state-based variability to provide more realistic performance predictions. The model builds on a Component Composition Language (CCL), which allows modelling component behaviour based on statecharts. The performance impact of a state is not further investigated, the focus of state modelling is directed on model checking of functional properties. Additionally, based on statecharts and certain behaviour claims, reliability of the system can be verified. Similarly, a state is modelled in the Component-Based Modelling Language (CBML) with the possibility to statically configure component parameters.

In the ProCom component model [29], designed for embedded systems, the state is modelled only statically with a set of component parameters. Further, the COMQUAD component model [24] is using Petri nets as a system behaviour model, the dependence of a service call on input data is however omitted. A number of other prediction models claim an ability to express state changes, but in many cases, they refer to behaviour protocol checking [15], state changes monitoring [26] or performance annotations based on measurements [6].

The *formal specification* model for testing of performance and reliability introduced by Hamlet et al. (i.e. HAMLET) [11] suggests to model a state as an additional input (additional floating point external variables loaded at the time of component execution) and provide tests showing functional aspects of a state. The *measurement* approach called AQUA [8] inherently monitors state impact (component description is given by the specification of Enterprise Java Beans, EJBs [33]) and showed how important it is to understand how a system state is interpreted. Another approach to measure EJB applications, developed by Gorton et al. (i.e. NICTA) [23] uses benchmarking methods to get platform-independent information, such as thread pool size, etc. The *simulation-based* approach MIDAS [1] 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 Sect. 3.1, we decided to extend the Palladio Component Model (PCM) [5] with further capabilities to model state-related information. The advantage of this model over the others is its clear component-based nature, already partial support for state modelling and the possibility to model usage profile in detail.

**Table 2** Component performance models comparison

| Component model | Design-time prediction models | | | | | | Formal specification methods | Measurement methods | | Simulation methods |
|---|---|---|---|---|---|---|---|---|---|---|
| State category | CB-SPE | PALLADIO | PECT | CBML | POCOM | COMQUAD | HAMLET | AQUA | NICTA | MIDAS |
| *Component-specific* | | | | | | | | | | |
| (a) Protocol | n/s | Prov./req.protocols | Statecharts | CBML | n/s | Petri-nets | n/s | n/s | n/s | n/s |
| (b) Internal | n/s | n/s | n/s | n/s | n/s | n/a | n/a | n/s | n/s | n/s |
| (c) Allocation | RT-UML PA profile Annotation | Static parameter, passive resources | n/s | n/a | Static parameter | n/s | n/a | Static parameter | Static parameter | Static parameter |
| (d) Configuration | | | n/s | Static parameter | | n/s | Additional input | | | |
| *User-specific* | | | | | | | | | | |
| (e) Global | n/s | n/s | Statecharts | CBML | n/s | n/s | n/a | n/s | n/s | n/s |
| (f) Allocation | n/s | Static parameter, passive resources | n/s | n/s | n/s | n/s | n/s | n/a | n/a | Static parameter |
| (g) Configuration | n/s | | n/s | n/s | n/s | n/s | n/s | n/a | n/a | |
| *User-specific* | | | | | | | | | | |
| (h) Session | n/s | Input data | n/s | n/s | n/s | n/a | Additional input | Static parameter | n/a | n/a |
| (i) Persistent | n/s | n/s | n/s | n/s | n/s | n/s | n/s | | n/a | n/a |

*n/s* Not supported, *n/a* information not available

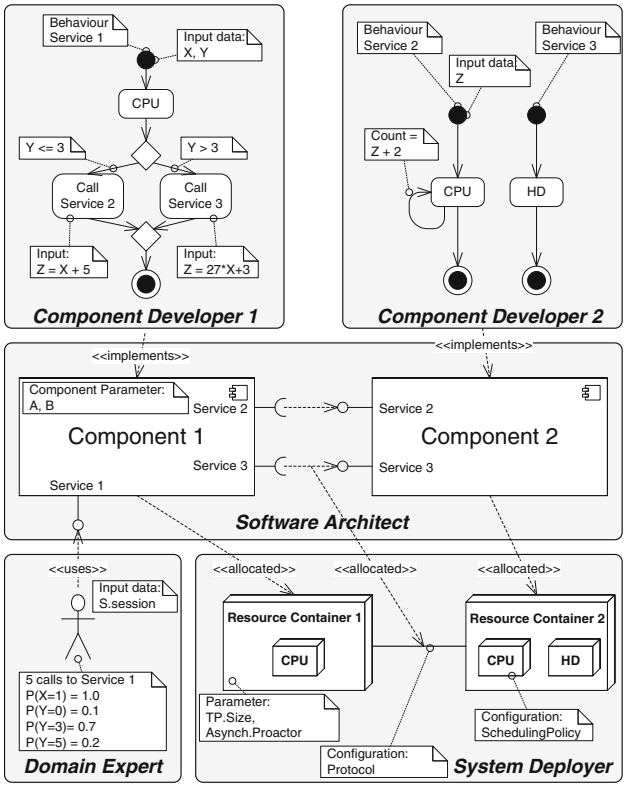



**Fig. 1** PCM example

The PCM is a modelling language specifically designed for performance prediction of component-based systems, with an automatic transformation into a discrete-event simulation of generalised queuing networks. Its available tool support (PCM Bench) allows performance engineers to predict various performance metrics, including the discussed response time, throughput and resource utilisation. All three properties are reported as random variables with probability distribution over possible values. The response time is expressed in given time units (e.g. seconds), throughput in number of service calls or data amount per time unit (e.g. kilobytes per second), and resource utilisation in the number of jobs currently occupying the resource.

First, the related foundations are introduced. Figure 1 shows a condensed example of a PCM instance. It consists of four models created by four developer roles in a parametric way, which allows the models to be updated independently of each other. 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.

*Component developers* are responsible for the specification of components, interfaces, and data types. Software components are the core entities of the PCM. Basic components contain an abstract behavioural specification called Service Effect Specification (SEFF) for each provided service. SEFFs describe how component services use resources during component-internal processing (via internal actions) and call required services (via external call actions) using an annotated control flow graph similar to UML Activity Diagrams. For performance prediction, component developers need to specify the demands of internal actions to resources, like CPUs or hard discs (see Fig. 1, action with resource demand on a CPU or a hard disc are called "CPU" or "HD"). Component developers can annotate external calls as well as control flow constructs with parameter dependencies. This allows the model to be adjusted to different system-level usage profiles. Parameter values can be of different types (e.g. String, Integer, Real, Composite) and can be characterised with random values to express the uncertainty while modelling large user groups. Furthermore, each component (or composed component) can define static component parameters. In the PCM, *parameter characterisations* [21,22] abstractly specify input and output parameters of component services with a focus on performance-relevant aspects. For example, the PCM allows defining the `VALUE`, `BYTESIZE`, `NUMBER_OF_ELEMENTS`, or `TYPE` of a parameter. The characterisations can be stochastic, e.g. the byte size of a data container can be specified by a probability mass function:

$$\texttt{data.BYTESIZE} = \texttt{IntPMF}[(1{,}000;\ 0.8)(2{,}000;\ 0.2)]$$

where `IntPMF` is a probability mass function over the domain of integers. The example specifies that `data` have a size of 1,000 bytes with probability 0.8 and a size of 2,000 with probability 0.2. *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 SEFFs 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 a state: (i) static component parameters (or properties) characterise the state of a component in an abstract and static way and hence offer a more flexible

parametrisation of the model. These parameters are propagated through development process differently, they are defined and initialised by a *component developer* and cannot 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 allow expressing a session state. Table 2 illustrates the capabilities of PCM to model the 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 a component internal state as described in Sect. 2.1(b). The PCM with this stateful extension is the first approach that supports the modelling of the internal state explicitly. With this extension, also a system-specific global state [Sect. 2.2(e)] can be modelled by adding a blackboard component that makes its internal state available to other components in the system, and the protocol state [Sect. 2.1(a)] modelled with a protocol-state holder stored within an internal-state parameter. Furthermore, component- and system-specific allocation and configuration states [Sects. 2.1(c), (d), 2.2(f), (g)] can be modelled with static component parameters. Finally, user-specific session [Sect. 2.3(h)] and persistent [Sect. 2.3(i)] states can be modelled with an additional input parameter in the usage model, and a persistent data store involved in the latter.

Only two additions to the PCM metamodel are required to model a component internal state. First, we declare a set of state variables for a component. Only the declared state variables can be used within a SEFF. Second, we add a `SetStateAction` to the SEFF, which allows us to set a 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. The state variable can be used in branch conditions or resource demands as a parameter. The use of *PCM Stateful* extension is illustrated in Sect. 6.

Figure 2 illustrates the PCM extension. Assume a component processing data. It performs a clean-up task after each megabyte of processed data. Thus, it keeps track of the amount of data processed. In the model, we store the limit of 1 MB in a component parameter named `dataLimitInMB.VALUE`, defining the component configuration state. We declare a state variable `processData.VALUE` and initialise it with the value 0, defining a component internal state. The SEFF of the component is shown in a state-chart-like notation in the figure. First, we modelled a `SetStateAction` to add the currently processed amount of data (available as `inputData.BYTESIZE`) to the `processData.VALUE` variable. Then, the data is processed in the `InternalAction` process. We omitted the resource demands for brevity. After processing the data, we check whether a clean-up is required in the `BranchAction`. If `processData.VALUE >= dataLimitInMB.VALUE`, we do the clean-up of 1 MB and set the state back to `processData.VALUE - dataLimitInMB.VALUE`. The second branch is empty.

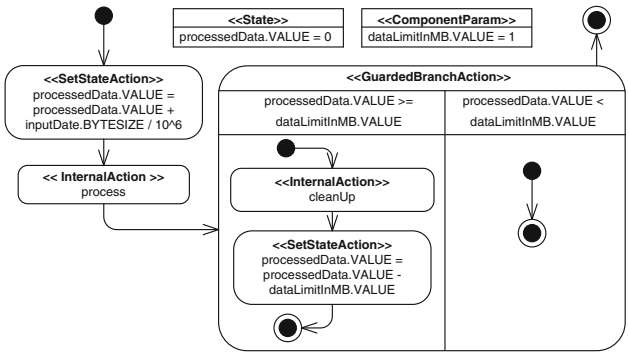



**Fig. 2** Stateful SEFF example

An extended PCM model can be analysed with the extended version of the *SimuCom* simulation presented in [5] 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 expressive 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 other 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 detect problems in a software design.

For example, assume a system service that becomes more expensive as more requests are served. Then, the response time of the system will ever increase ("The Ramp" antipattern [30]) and no steady state can be reached. With the extended state modelling, this performance antipattern can be detected based on the simulation results.

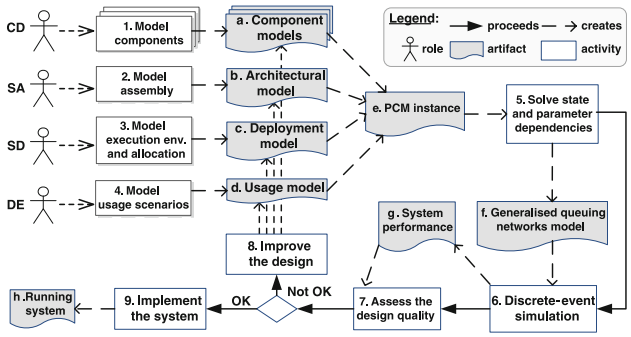



**Fig. 3** PCM process outline

## 4 Outline of the approach

After identifying the types of state-related information in component-based systems, and extending the PCM performance prediction with state-modelling constructs, this section together with Sect. 5 elaborates the main contribution of this paper—a study of the performance impact of the identified state categories and an analysis of the influences that should drive the decision on the abstraction level of state modelling.

Within the PCM process of performance prediction in Fig. 3, the new state-effect analysis aims to support the activities 1–4 and 8, which are the only manual activities in the design process, performed by the four developer roles discussed in Sect. 3.2. We introduce and validate eight state-modelling heuristics aimed to guide *component developers* (*CD*) in refining the component models with component-specific states, *software architects* (*SA*) in refining the architectural models with system-specific states, and *domain experts* (*DE*) in refining the usage models with user-specific states.

In design-time performance prediction, the issue of effective model refinement with an additional information has already been addressed for many system details, including service parameters, return values, or usage-profile information. Our approach gives an insight into the issue of state modelling, which has not been addressed so far, and tries to help software engineers to find the balance between accuracy and complexity of models more competently.

In particular, the approach aims at helping software engineers to decide if the increase in the prediction accuracy introduced by the state modelling outbalances the price that needs to be paid for the increased model complexity and size. To compare the two metrics, we first discuss the quantification of the performance impact (see Sect. 4.1) and the model-size cost (see Sect. 4.2). Second, we discuss the similarities among the state categories with respect to performance impact and model-size cost (in Sect. 4.3), and design four classes clustering the state categories that are similar with respect to these characteristics (hence the same best practices apply to their modelling). Each of the classes is later analysed in Sect. 5. For each class, we discuss the observations about its performance impact and model-size cost and design a number of best practices (heuristics) condensing the advices for the software engineers with respect to state modelling. Each heuristic is experimentally evaluated and the results of the evaluation are summarised in the text. Further details of the evaluation are included at the website of the approach [17].

### 4.1 Quantification of performance impact

With the adopted PCM, software architects can quantify three aspects of system performance: *response time* (of a component or system service), *throughput* (of a service or communication link), and *resource utilisation* (of a hardware resource). All the three metrics are reported as random variables with probability distribution over possible values. The response time is expressed in given time units (e.g. seconds), throughput in number of service calls or data amount per time unit (e.g. the number of transactions per second), and resource utilisation in the number of jobs currently occupying the resource.

The three performance metrics for all the individual model elements are propagated to the *system response time*, which quantifies the response time of a given usage profile. The system response time is dependent on the response times of the user-called system services, which depend on the response times of component services included in the triggered control flow, resource utilisation of the system hardware resources employed during service execution, and throughput of the utilised communication links (due to contention and overloading effects).

In this paper, we are hence interested in the impact of state modelling on the system response time, which can be quantified with different focus, including best/worse time, mean time, median or variance. In addition software architects can be interested in the probability distribution or the time series of possible times. Since for each of the stateful abstractions, the state-modelling advices or heuristics may have different validity, we discuss all of them in Sect. 5 although we focus on the probability distribution, which is the default metric used in the PCM and most of the mentioned metrics can be derived from the distribution function.

The individual metrics of system response time discussed in this paper are: mean value, median value, best/worst case, variance, probability distribution, and time series. The first two metrics, the *mean* and *median* values, approximate the expected system response time. Although both are very popular in statistics, they are usually considered too coarse-grained for performance engineering. The *best* and *worst case* values are given to make the response-time characterisation more detailed and are very important in

predictions for safety critical systems. Together with the *variance*, these metrics already characterise the possible response-time values quite concisely. If all these information are needed, the full *probability distribution* of the response-time values is often required, alternatively formulated as a *cumulative probability function*. The most detailed metric is the *time series* which reports possible response-time values of individual system services in connection to the time when the service execution has been started. The *time series* provide a view on the evolution of the response time over the time, which is a basis for transient analysis of systems, although *time series* are the most difficult metric to analyse.

### 4.2 Quantification of model complexity

The complexity of a model can be best understood when the model is translated to a low-level formal language with clearly defined size. Labelled transition systems are one of the formalisms commonly employed for this purpose. In the case of component-based systems, different kinds of interacting automata [37,38] are employed to specify large labelled transition systems via composition of automata-based models of individual components. In [37], the inclusion of a state-related 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 modelling 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 also of the models of all state attributes (whose size corresponds to the number of possible state values).

Since the size of a composite component-interacting 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 size of the initial stateless model with the size of the state model. However, our experience shows that this case is very unlikely to occur, and that the stateful model can be even smaller than the initial stateless model, due to a higher certainty about the future behaviour of the system. A more detailed discussion of this phenomenon can be found in the sub-sections of Sect. 5.

### 4.3 Diversity of state categories

In Sect. 2, we have identified nine state categories present in component-based systems and classified them along two dimensions. While the classification is valid for any quality model of a component-based software system, i.e. independent of performance and the accuracy of its prediction, this section summarises the performance-related similarities and differences of the state categories observed during our experiments. In particular, once we focus on the characteristics of *performance impact* and the increase in *model complexity* (i.e. *model-size cost*), as defined in Sects. 4.1 and 4.2, strong similarities between some categories can be observed, which suggest that the same best practices for their inclusion in performance models will apply.

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

System vs. component-specific state: Even if the component-based system behaviour is structured into components (building system architecture), its core is in the interaction of component 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: From the point of view of performance-prediction models, the persistent state is analogous to a session state for one life-lasting session, and hence holding no significant characteristics distinguishing the two.

The identified similarities cluster the defined state categories into four classes: (1) protocol state, (2) internal and global state, (3) allocation and configuration state, and (4) session and persistent state. Since the same best practices on state modelling (i.e. heuristics suggesting if the state should be included in the model) apply to all state categories within the same class, the next section relates the introduced heuristics to the four classes only.

## 5 State-effect analysis

For each of the four state classes identified in Sect. 4.3, this section discusses the performance impact and model-size cost of state modelling and designs a set of modelling heuristics. The heuristics are evaluated on a set of experiments and the conclusions from the evaluation discussed in the text.

The relative increase in performance accuracy and model complexity is quantified through a comparison of a stateful PCM model to a stateless model of the same system, where the state-dependent decisions are abstracted probabilistically. The discussion of performance impact is moreover refined with respect to a number of response-time metrics defined in Sect. 4.1.

### 5.1 Protocol state

The protocol state, which is the only state category included in this class, is used for a very specific purpose. It holds

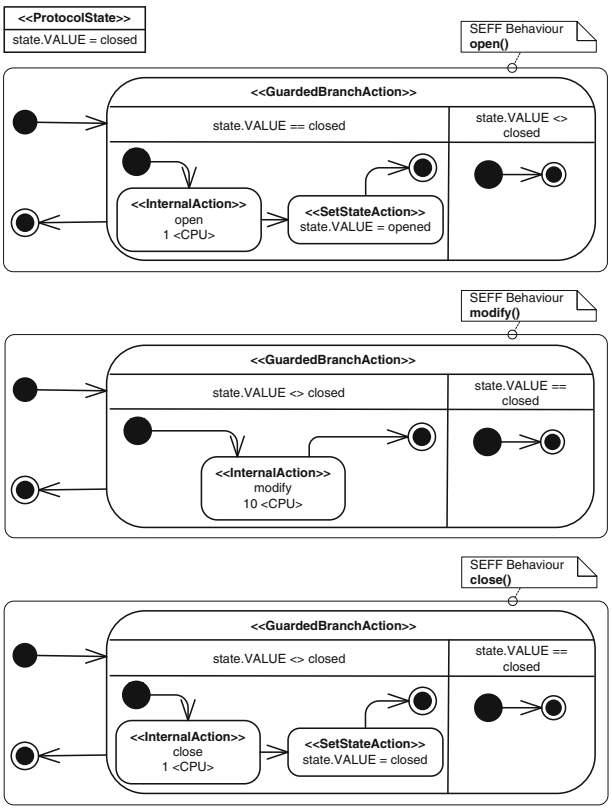



**Fig. 4** A Protocol State example

the information about currently acceptable service calls of a component.

*Example*: Recall the protocol-state example outlined in Sect. 2 with two state values: *closed*, when the only acceptable service call is `open()`, and *opened*, when the component can accept calls on `modify()` and `close()`. The stateful PCM model of the example in Fig. 4 consists of three SEFF models. Each SEFF starts with a branch condition deciding if the service will be executed or the service request ignored. While in the stateful model, the branches are guarded with the current value of the protocol state, updated after the execution of `open()` and `close()`, the probabilistic-abstraction model would set fixed probabilities to the branches based on the expected likelihood of the alternatives.

### *5.1.1 Performance impact*

Observations: A number of performed experiments with different variations of the probabilistic model revealed two main observations on the accuracy of the stateful model comparing to the stateless (i.e. probabilistic-abstraction) model.

Observation 1: 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 typically defined independently by different developer roles.

Observation 2: Even if the usage profile is known, the actual probabilities of service execution depend on component's environment through which the usage profile is propagated, and thus can be very hard to quantify.

Heuristics: There are two heuristics that can be derived from the observations.

Heuristic 1.1: *The importance of protocol-state modelling raises with lower knowledge of the usage profile.*

Experimental evaluation: Our experiments have shown that already a very little inaccuracy in the usage profile may lead to a very imprecise stateless (i.e. probabilistic-abstraction) model, since the inaccuracies can be easily magnified by system control flow.[1] While any user input is in the stateful model that readily propagated to the corresponding system state (valid for the *protocol*, *internal* or *global* state in particular), in the stateless model, the input effects may be distributed throughout the whole system model. There are two performance-related arguments that justify the inclusion of a state-dependent parameter into the model in this case. Firstly, significantly more effort is required in the stateless model to update its transition probabilities to a more accurate usage profile. Secondly, an adaptation of the probabilities in the stateless model does not need to be sufficient to reflect the usage-profile change accurately. A structural change of the model may be necessary (valid for all response-time metrics). To illustrate these two phenomena, we have experimented with an explicit (i.e. state-free) propagation of usage profile changes into the system model, applying it to all the protocol, internal and global state. To save space, we present the evaluation details in Heuristic 2.1, which is an analogy of Heuristic 1.1 for internal/global state.

Heuristic 1.2: *The importance of protocol-state modelling raises with higher complexity of the component's environment.*

Experimental evaluation: There are situations common in complex systems, when it may be very difficult or even impossible to estimate the probabilities in a stateless model precisely. A simple exemplary model illustrating this phenomenon can be built on the fact that the probabilistic abstraction can hardly be foreseen in the models where the same service is called twice and each time behaves differently based on the actual protocol-state value that may change in the meantime. In particular, consider an extension of the example in Fig. 4 with an additional state value *active* and service `activate()`. A schema of the

[1] Note that the PCM supports value-passing and value-guarded control-flow constructs, which imply that already a minor modification of an input value in the usage profile may influence system behaviour significantly.

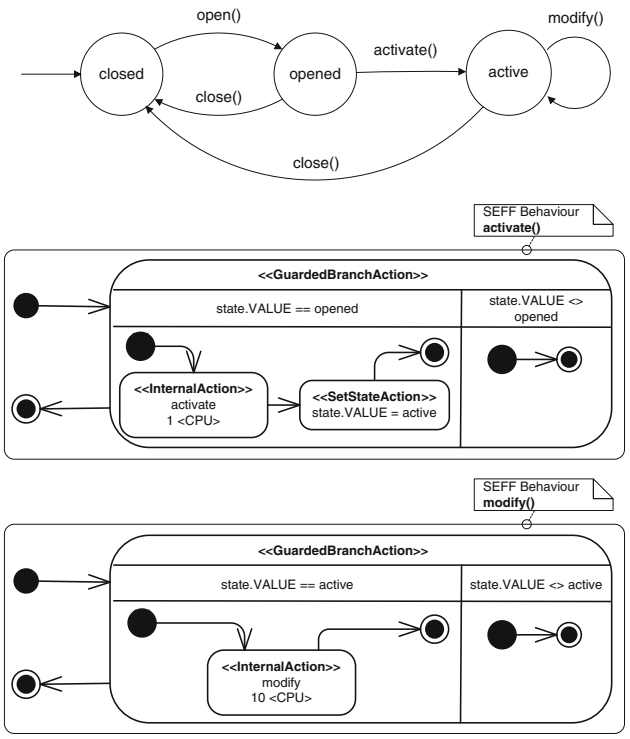



**Fig. 5** An extended protocol state example

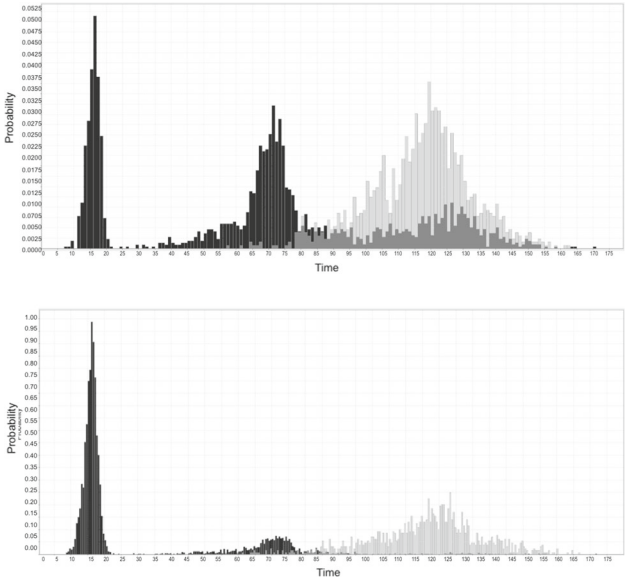



**Fig. 6** Stateful (*light*) vs. probabilistic (*dark*) results

state values is shown in Fig. 5, together with the model of `activate()` and updated model of `modify()`. The experimental usage model considers a simple usage profile with a single scenario (for 10 users in the system) calling a sequence: `open(); modify(); activate(); modify(); close()`. While in the stateful model, the executability of each service is given by the actual value of the protocol state, for the probabilistic model, we need to estimate the transition probabilities for all the branches in the SEFFs, which are trivial for `open()`, `activate()` and `close()`, but nontrivial for `modify()`. Figure 6 illustrates a typical comparison of stateful (light) vs. stateless (dark) model results for two variations of this example (increased complexity of the environment), showing that the deviation affects all the studied response-time metrics.

### *5.1.2 Model-size costs*

The stateful model of each service connected to a protocol state has a unified form, having two independent alternatives: the first (complex) if the service shall be executed, and the second (trivial) if the call shall be ignored (see Fig. 4). In a stateful model of the service, two sources of model-size increase can be observed.

Observation 1: *An increase due to state update after service execution.* The model size increases with the number of state updates after service execution. The increase in this case is however negligible.

Observation 2: *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 under more than one value 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.

## 5.2 Internal/global state

The internal state, as well as the global state, holds local (resp. global) information used to coordinate the behaviour of the system or its components.

*Example*: Consider an example of a `processData()` service outlined in Fig. 7, with the internal-state parameter *processed* storing the amount of processed data, and coordinating the component to either process additional data or perform cleanup. Besides, consider an example of a very simple library search functionality in Fig. 8, which employs two global states $X$ and $Y$ to store search parameters shared by the `findBook()` and `archiveSearch()` services. In this example, the two global states imitate value propagation in the system [(from the user to `findBook()` and later to `archiveSearch()`]. In both examples, the probabilistic model would be analogous to the stateful model, with the

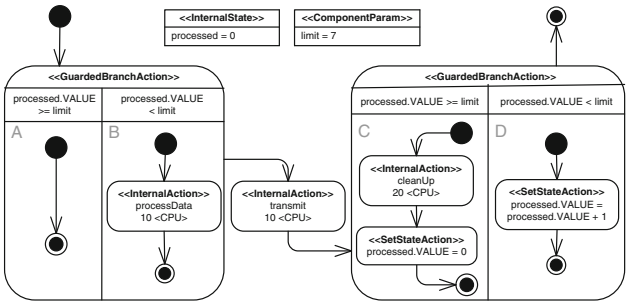



**Fig. 7** An internal state example of a `processData()` SEFF

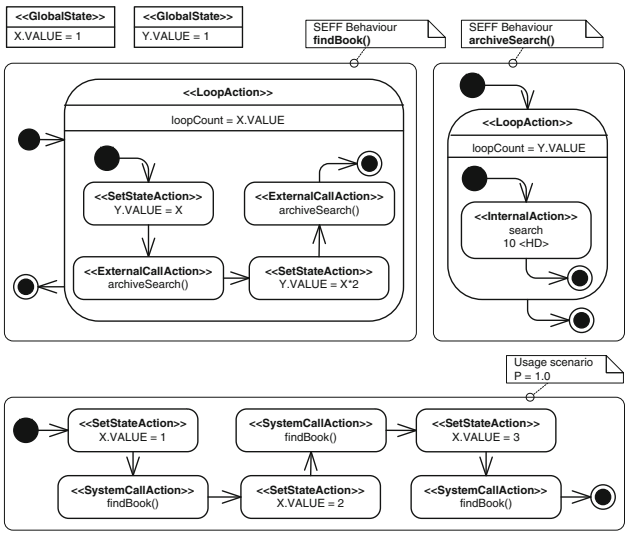



**Fig. 8** A global state example of library search

branches and loop counts guarded with the probabilities of state values.

### *5.2.1 Performance impact*

Observations: Besides the observations defined by Heuristics 1.1 and 1.2, which are valid for this state category as well, the example in Fig. 7 discloses an additional influencing factor specific to this state category. It is connected to a possible correlation of state values in subsequent branches (or other control-flow decisions) guarded with an internal/global state.

Observation 1: The performance impact of the internal/global-state modelling highly depends on the a priori knowledge of the usage profile and the complexity of the system (environment of each studied component).

Observation 2: Recall the example in Fig. 7 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

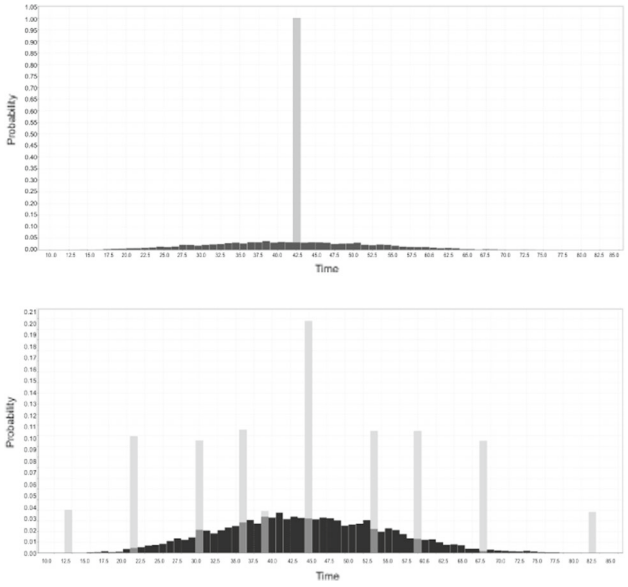



**Fig. 9** Stateful (*light*) vs. probabilistic (*dark*) results

B followed by D), in the probabilistic model, four alternatives are possible (both A and B can be followed by both C and D).

Heuristics: The observations can be summarised with the following heuristics.

Heuristic 2.1: *The importance of internal/global-state modelling raises with lower knowledge of the usage profile.*

Experimental evaluation: Our experiments revealed that already a very little inaccuracy in the usage profile may lead to a very imprecise stateless (i.e. probabilistic-abstraction) model, since the inaccuracies can be easily magnified by system control flow. This observation is valid for three types of states (due to their usage dependence): the protocol, internal and global state. For space reasons, we demonstrate the observation on a single global-state example of a simple library search in Fig. 8. In the evaluation, we have first studied the effects of usage profile propagation to the control flow (see [17] for more details) and then designed a number of usage scenarios validating the observation.

In Fig. 9, we present two graphs comparing the stateful (light) and probabilistic-abstraction (dark) model results, where the top graph corresponds to the usage scenario in Fig. 8 and the bottom graph to a similar usage scenario where each change of the *X.VALUE* is uncertain (equals to 1 in 34 % of cases, 2 in 33 % of cases and 3 in 33 % of cases). The difference in the results clearly demonstrates higher variability of the probabilistic results, which is with increasing complexity of the system propagated also to the deviation in the mean and median values.

Heuristic 2.2: *The importance of internal/global-state modelling raises with higher complexity of the system (environment of each component).*

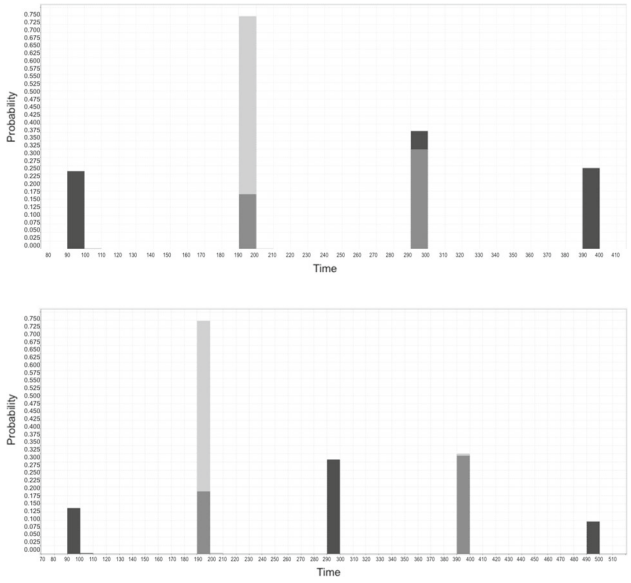



**Fig. 10** Stateful (*light*) vs. probabilistic (*dark*) results

Experimental evaluation: The evaluation of Heuristic 2.2 can be built on analogous reasoning to Heuristic 1.2. Two already presented models can be used for demonstration of the observations. First, the model employed in Heuristic 1.2 could be modelled with an internal state and used here (see Figs. 4, 5, 6). Second, the model employed in Heuristic 2.1 integrates also the complexity of the environment and demonstrates this observation (see Figs. 8, 9).

Heuristic 2.3: *The importance of internal/global-state modelling raises with higher correlation of subsequent state-driven decisions.*

Experimental evaluation: In the experimental evaluation, we used a number of models analogous to the example in Fig. 7, with a sequence of two or more state-dependent branches and internal actions in between. The experiments (see Figs. 10, 17 for more details) show that even if the probabilities of the branches accurately reflect the usage profile, the results computed from the stateless models can be very imprecise. Already in very simple models (one service with two or three branches), the probability distribution (mainly the variance and best/worst case) of the stateless-model results deviates significantly from the stateful-model distribution (see Fig. 10). The mean and median values tend to be quite stable for these simple examples, and start to deviate when more complexity is introduced into the models (see Figs. 15, 16 in an analogous Heuristic 4.2, where the branches are placed in a loop).

#### *5.2.2 Model-size costs*

The model of a service involving an internal/global state can have much more variability than in the case of a 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 likelihood is decreased by the factors summarised by the following observations.

Observation 1: *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 (analogously to the argument for the protocol state).

Observation 2: *A low number of independent state-guarded branches.* Recall the example in Fig. 7. While in the stateful model with dependent branches two behaviours were possible (A,C and B,D), if the branch conditions were independent, four behaviours would be possible (A,C; B,C; A,D and B,D). However, a high number of independent branches does not increase the number of vertices in the model. It only increases the number of transitions and hence the number and complexity of behaviour-describing paths.

Observation 3: *A small number of state updates.* A smaller number of state updates imply a higher likelihood that some of the branch conditions will always (or at least often) be evaluated as false and the behaviours that follow them will not repeat often in the model (or will not appear at all).

Observation 4: *A limited scope of the state.* The bigger part of the system is independent of the state value, the smaller model-size increase can be expected. That is, the model-size increase tends to be significantly smaller in the case of component internal states compared to the global states.

### 5.3 Allocation/configuration state

This class comprises four state categories, in particular the component-specific and system-specific allocation and configuration state, all coordinating system behaviour according to a fixed (deployment or configuration) parameter.

*Example*: Consider a simple reference example in Fig. 11 executing a certain functionality in a sequence of loops with the loop count dependent on an allocation state defining the maximal queue length within the system. The most important property shared by the four state categories represented by this class is that they are fixed before the execution (in Fig. 11 modelled at the beginning of the usage scenario) and hence coordinate component or system behaviour in a unified way during system execution. Again, while in the stateful model, branches may be guarded with state values, in the stateless model, the same branches are guarded with probabilities (reflecting the likelihood of possible parameter values). If we are uncertain about the actual value also in the stateful model, we can include this uncertainty into the

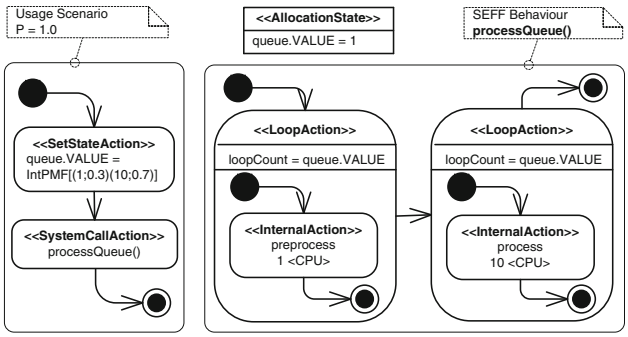



**Fig. 11** An allocation/configuration-state example

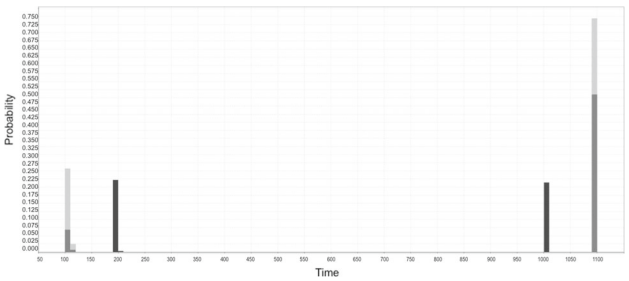



**Fig. 12** Stateful (*light*) vs. probabilistic (*dark*) results

usage profile (in Fig. 11 the value of *queue* is configured to 1 in 30 % of cases and 10 in 70 % of cases), which before triggering the system execution configures the parameters with the corresponding probabilities of their values and then uses them in a fixed way along system execution. On the other hand, if we have an absolute certainty about the value of the parameter, we can reduce the stateless model (and actually also the stateful model) to keep only the branch behaviours conforming to the actual value of the parameter.

### *5.3.1 Performance impact*

Observations: The above-mentioned specifics imply two main observations influencing the effect of allocation/configuration-state modelling.

Observation 1: As distinct from so far discussed categories, the general influence of the allocation/configuration state to system performance is independent of the usage and the environment. For each service, the state-guarded branches are evaluated in a fixed way, irrespective of the service clients.

Observation 2: On the other hand, the prediction accuracy is highly dependent on the knowledge of deployment/configuration parameters, which allow the architect to cut off the behavioural branches in the stateless model that go against the expected value of the parameter. When this information is not available to the component developer (since it is determined by a different role) and the uncertainty about the state value needs to be expressed with probabilities, the probabilistic model exhibits high inaccuracies.

Heuristics: The following heuristic can be derived from the observations.

Heuristic 3.1: *The importance of allocation/configuration-state modelling raises with lower knowledge of deployment/configuration parameters.*

Experimental evaluation: The experimental evaluation reveals that whenever there is any uncertainty about the value of the parameters (as illustrated in Fig. 11), which need to be in the stateless model modelled with probabilities, the model may become very imprecise. The reason for this fact is that while in the stateful model, the parameter value for the whole system execution remains the same (the uncertainty about the parameter value is included in only one place, in the usage profile before triggering system execution), the stateless model includes also the behaviours reflecting the unrealistic cases of parameter changes during system execution (i.e. 1 loop in the first *LoopAction* followed by 10 loops in the second one or vice versa in Fig. 11). The deviation of the stateless model from the stateful results tends to exhibit a common phenomenon regarding the probability distribution of the reported values. In particular, while the mean and the median of the results used to be the same (or very similar), the variance of the stateless results tends to be higher, with more possible values in the stateless case. See Fig. 12 for results of example in Figs. 11, and 17 for a more detailed discussion.

### *5.3.2 Model-size costs*

The costs of state modelling in terms of the model size are influenced by the following observations about the expected size of the stateful and stateless model for the same system (or system element).

Observation 1: *A stateful model of a single system element uses to be larger than the stateless model of the same element.* This is the case whenever the architect of the stateless model cuts off those branch behaviours that go against the expected value of the parameter.

Observation 2: *In the case of intended model reuse, the stateful model of a single system element often has the same size as the stateless model of the same element.* If we do not know the value of the deployment/configuration parameter in advance (typical in the case of model reuse in different contexts), the stateless (probabilistic abstraction) model needs to include behaviours implied by all possible parameter values, and hence has the same complexity as the stateful model.

Observation 3: *System models resulting from the composition of individual stateful model elements are not larger than the stateless composite system models.* As the value of the allocation/configuration state does not change during system

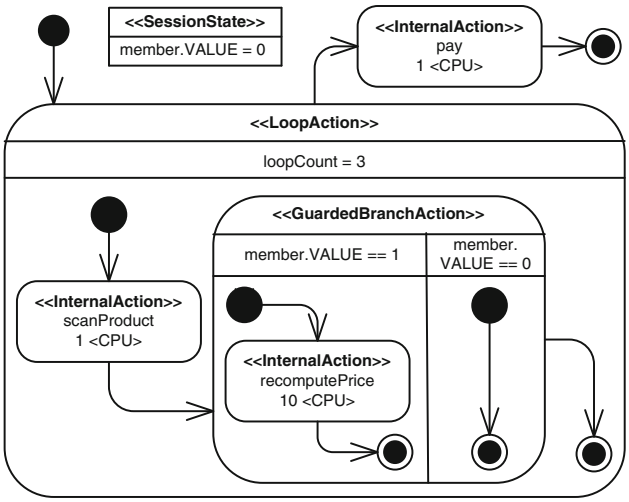



**Fig. 13** A session/persistent-state example

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 can even be smaller than in its probabilistic variant.

### 5.4 Session/persistent state

The session state, as well as the persistent state, holds an information remembered for each individual user, and used to customise system behaviour accordingly.

*Example*: Consider a session-state example in Fig. 13 with sessions connected to individual sales in a supermarket, parametrised by information about the customer (has a club member card or not). In the example, each product scan (3 in total) can be followed by recomputation of the product price (if the customer is a club member), which may significantly influence system performance for the two types of customers. The PCM model can be very simple, propagating the user-specific state in terms of an input value throughout the whole session. In the stateless model, the state-guarded decisions are again replaced with probabilistic decisions. Any uncertainty about the parameter value can be expressed analogously to Sect. 5.3.

#### *5.4.1 Performance impact*

Observations: The session/persistent state exhibits some similarities, but also differences to all the state classes discussed above. It is very similar to the allocation/configuration 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 from the protocol state but similarly to the internal/global state. This implies the following two observations.

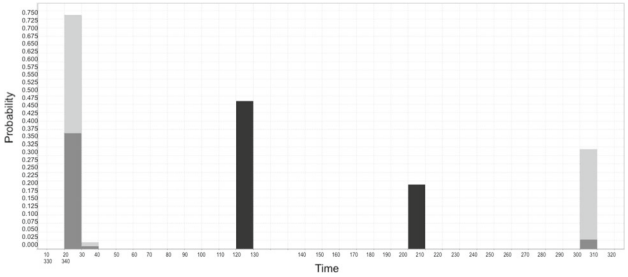



**Fig. 14** Stateful (*light*) vs. probabilistic (*dark*) results

Observation 1: The impact is not very dependent on the usage profile and environment, but highly dependent on the knowledge of the distribution of the state values (similarly to the knowledge of deployment parameters in the case of the allocation/configuration state).

Observation 2: Since the subsequent queries on the state value are highly correlated, probabilistic models can hardly model session/persistent-state dependent behaviour faithfully (similarly to the internal/global state).

This state class hence plays a significant role in the model, due to the implied strong correlation of subsequent state-guarded branches, and changeability of the state value along system execution.

Heuristics: There are two heuristics that can be derived from the observations.

Heuristic 4.1: *The importance of session/persistent-state modelling raises with lower knowledge of the corresponding user-given parameter values.*

Experimental evaluation: Consider the sale example in Fig. 13, where the value of the *member* state is uncertain, reflecting that 30 % of customers are club members and 70 % are not. Similar to Heuristic 3.1, the experimental evaluation reveals that whenever there is any uncertainty about the value of the session/persistent state, the results computed from the stateless model deviate from the accurate results of the stateful model (with respect to all the studied response-time metrics—and mainly the probability distribution). The results for our simple sale example are in Fig. 14.

Heuristic 4.2: *The importance of session/persistent-state modelling raises with higher correlation of subsequent state-driven decisions, which is typically very high.*

Experimental evaluation: The evaluation was built on a set of examples derived from the sale example in Fig. 13, where the number of loop counts (i.e. the number of subsequent state-dependent branches) increases from 3 (in Fig. 14) to 6 and later 9 (both in Fig. 15). Moreover, Fig. 16 combines the loop counts into one example where in 30 % of cases the loop count is 3, in 50 % of cases the loop count is 6, and in 20 % of cases the loop count is 9, to demonstrate that the complexity of the system further deepens the deviation of the stateless and stateful results.

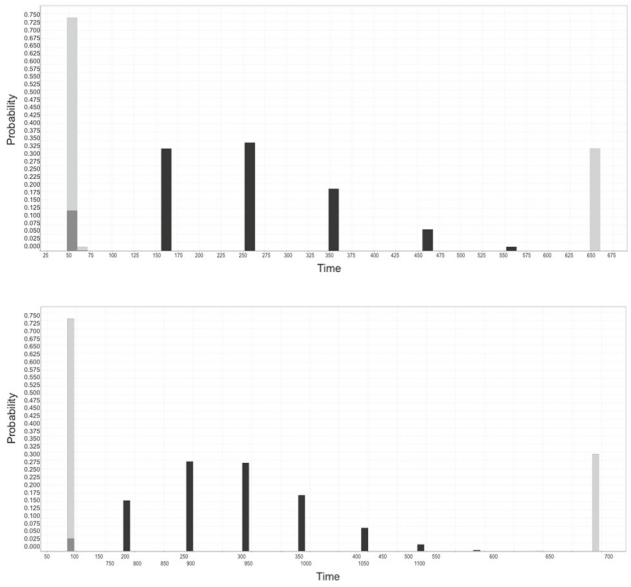



**Fig. 15** Stateful (*light*) vs. probabilistic (*dark*) results

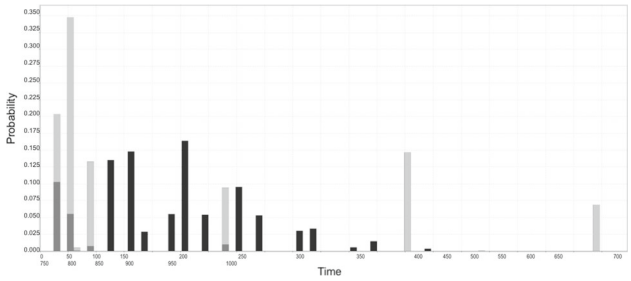



**Fig. 16** Stateful (light) vs. probabilistic (dark) results

The experiments (with different loop counts) demonstrate that the deviation affects all the response-time metrics. Moreover, the practical performance impact is strengthened by the fact that the correlation of subsequent state-driven decision is likely to be very high, since the state value (for both the session and persistent state) is highly stable along system execution (i.e. also between the state-dependent decisions).

### *5.4.2 Model-size costs*

The experience learned about the size of the model can be summarised by the following observations.

Observation 1: *Connection of component behaviour to the state value.* The increase due to remembering the actual state value is dependent on the connection of component behaviour to the state value, similarly to the internal/global state. The weaker the connection is, the closer the model can grow to the worst case.

Observation 2: *Correlation of subsequent decisions.* Thanks to the correlation of subsequent decisions (branches and loops), there is basically no complexity increase in terms of the behavioural paths.

Observation 3: *State update.* There is basically no size increase due to state update, since the state is not updated inside the system, and occurs very rarely.

## 6 Validation

The goal of this section was to evaluate the influence of stateful dependencies on the performance of real-world applications. We decided to use the *SPECjms2007 Benchmark* [31] for evaluation due to its particular challenge for our approach. SPECjms2007 is an official industry-standard benchmark for in-depth performance analysis of enterprise message-oriented middleware (MOM) servers based on Java Message Service (JMS). This benchmark is designed to measure the end-to-end performance of all components that make up the application environment, including hardware, JMS server software, JVM software, database software if used for message persistence, and the system network. Thus, we expect it to be tailored including relevant stateful dependencies and thus can be used to evaluate not only the systems in a steady state, but even to evaluate a transient nature of real-world enterprise systems.

In the following, we first evaluate stateful dependencies observed in the measurements on this benchmark. Second, based on the previously introduced state categorisation and heuristics, we propose a model in PCM integrating the state-related information for one of the identified stateful dependencies. Third, we validate accuracy of our model by comparing the prediction results using this model to the measurements of the real system. Moreover, we discuss resulting costs in simulation of the introduced model. To hide from the user, the model complexity resulting from the state introduction, we encapsulate state-dependent behaviour as parts of the automated model refinements, called *model completions* [18]. Model completions refine models using model-to-model transformations integrating complex subsystems, in this case, the subsystems represent modelled state-dependent behaviour. The main advantage of this automated refinement is the reuse of expert knowledge provided for performance engineers, who do not have to understand the complex stateful models and only have to provide initial information, which they have accessible (e.g. that certain model elements communicate by a transaction with certain size) to integrate automatically complex transactional subsystems in their models.

### 6.1 Evaluating stateful dependencies in the SPECjms2007 benchmark

The SPECjms2007 Benchmark emulates a supply chain of a supermarket company. The participants involved are the supermarket company headquarters, its stores, its distribution

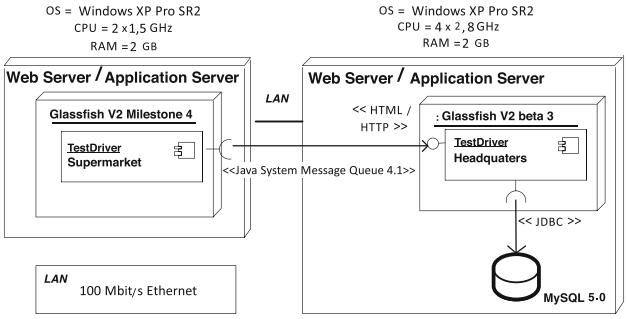



**Fig. 17** Experimental setup for performance measurements

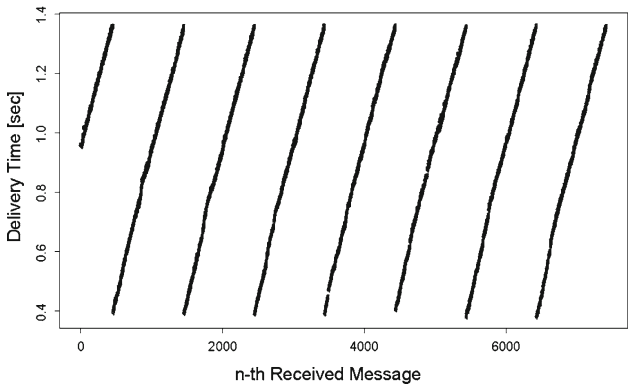



**Fig. 18** Time series of a transaction measurement with 1,000 messages per transaction set (one point stands for exactly one measurement of one message)

centres and its suppliers. The emulated scenario meets the requirements of realistic enterprise application, which is essential for the evaluation of our approach.

We identified a number of relevant stateful dependencies in the benchmarks scenario. The SPECjms2007 benchmark is based on a representative workload scenario that should represent a typical transaction mix. Supported features are chosen according to their usage in real-life systems. Such workload transaction mix can consist of transactional or non-transactional messages, persistent or non-persistent messages, durable or non-durable subscriptions or prioritised messages of specific type, size or origin.

To validate our approach, we evaluated three representatives covering the diversity of state categories as discussed in Sect. 4.3: (i) the *component internal state*, which extends the model of the *Message-Oriented Middleware* (*MOM*) to support transactions, (ii) the *configuration state*, which allows modelling the effects of cache utilisation on performance and (iii) the *persistent state*, which allows modelling a state-dependent load balancing strategies (i.e. round-robin replica assignment per request) among a number of replicas (e.g. replicas of components) in the software system. In addition to the case studies introduced in this paper, we conducted 16 simulation experiments described in more detail on the website of the approach [17].

Figure 17 shows the architecture and the setup used for evaluation. The setup comprises two nodes that are connected with a 100 MBit/s Ethernet. We deploy the test drivers for the benchmark implementation on a Glassfish V2 Web Server. The headquarters access a MySQL 5.0.95 database through JDBC. A dedicated experiment controller executes the measurements. Each run of the experiment initiated 1,000 messages and we measured delivery time of each message in nano-seconds. For the message persistence, we used the database. In the following, we discuss modelling of the first stateful representatives: the component internal state modelling transactions. We explain how the introduced stateful component model extension can be used to create more accurate prediction models on an example of transactional message-oriented communication in the benchmark scenario.

### 6.2 Component internal state: transactional message-oriented communication

Message-oriented communication among the participants is implemented using the JMS standard [12] and supports transactions for messages. The transactions guarantee that all messages are delivered to all receivers in the order they have been sent. To achieve this behaviour, Sun's JMS implementation [32] waits for all incoming messages of a transaction and then delivers them sequentially. This results in delivery times depicted in Fig. 18, showing the measured delivery times for a series of transactions with 1,000 messages each [the sender initiates a new transaction (as part of a session), passes 1,000 messages to the middleware, and finally, commits the transaction]. All messages arrive within the first 0.4 s and are delivered sequentially within the next second. This behaviour leads to delivery times of 0.4 s at minimum. The delivery times grow linearly until the transaction is completed.

This measurement illustrates that the position of a message in the transaction set (i.e. current state of transaction) 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, the prediction model aiming to predict observed state-dependent behaviour needs to keep track of the periodical utilisation of resources (e.g. CPU) and the messages that are part of a transaction, that influence system performance. To reflect this behaviour, a state-related modelling construct needs to be part of the performance model. Then the identification of idle periods between the transactions is possible, which eases the optimisation of the resource utilisation (e.g. load balancing strategies).

Message-oriented middleware (MOM): In this section, we use the stateful PCM introduced in Sect. 3.2.1 to create a required model allowing to predict the delivery time of

transactional messages as observed from the measurements of the real system in Fig. 18. Transactional messages are common in today's enterprise applications, such as implemented by SPECjms2007 Benchmark [31]. However, the transactions used in the supply chain management for supermarkets in the benchmark are limited to small, predefined transaction sizes. To provide a better evaluation, we implemented an application that allows configuring the number of messages sent 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 the model of MOM introduced by Happe et al. [13] in the following. The MOM subsumes several components that reflect the influence of different middleware configurations such as guaranteed delivery, competing consumers, or selective consumers (see Fig. 19). A model completion implemented as a model-to-model transformation (in QVT Relational [28]) generates the necessary MOM components and integrates them into a software architecture. We have already demonstrated that the model of MOM (stateless, thus without transactions) can predict the performance of a SPECjms2007 scenario with an accuracy of 5–10 % [13]. In the subsequent paragraphs, we present an extension of this model that enables the prediction of the transactions' effect on the delivery time of a message.

Stateful wrapper for transactions: In the previous model of MOM, the transactional delivery was not supported, because of the requirement of the underlying performance model to be as simple as possible. That does not allow modelling of stateful components because of the complexity issues that developer would have to deal with. We decided to extend the model of MOM in the way that the usage of stateful components will be hidden from the developer. By an introduction of stateful wrapper, which encapsulates state-dependent behaviour, the black-box principle of components would not be violated as the extended component itself remains unchanged. The stateful wrapper will be inserted into the model on demand by the model completion as an wrapper around previously stateless component. This wrapper will then manage calls to the methods of the component based on the state value.

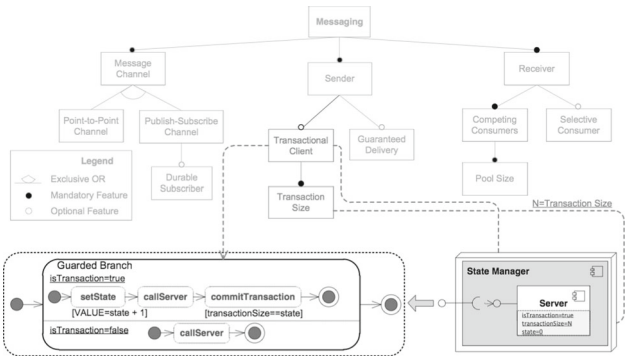



**Fig. 19** Model of MOM with the transactional delivery extension

Figure 19 illustrates simple structure of such wrapper to support transactions and mapping of model changes to the required features. The stateful wrapper around the component then manages incoming messages with respect to the configuration of the transaction, thus incoming messages are collected until the transaction size is reached and then committed for the transmission.

The stateful wrapper allows modelling specific state-dependent behaviour in the system. We can apply this approach to any component to model a particular resource-independent capacity constraint of a component, e.g. a throughput of requests dependent on the request's size. The wrapper component encapsulates and reflects the throughput restriction using the features that the extended PCM provides. Hence, it is possible to adapt how the component would behave at runtime. For example, one component can store the information on which queues are locked or which component has locked the queues. Such effects could be automatically detected in existing implementations of software systems in operation by automated measurements and experiments (e.g. by Software Performance Cockpit [14,35]), but it was not possible to model them at design time without the notion of state in prediction models; thus, these effects were not visible in performance predictions and were observed only late in operation. As a result, performance problems can be identified only late in the development and so the development costs increase. Using previously introduced heuristics, the type of the stateful wrapper to model the state dependence can be selected, modelled in the extended PCM and automatically inserted into the model using model completions. In the following, we discuss the details of the model resulting from the application of a stateful wrapper to extend the model of MOM.

Model completion for MOM with transactions: Figure 20 shows the components and connections that are part of the model (see [13] for details). The model consists of *adapter components* and *middleware components*. The adaptors forward requests and calls to 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`

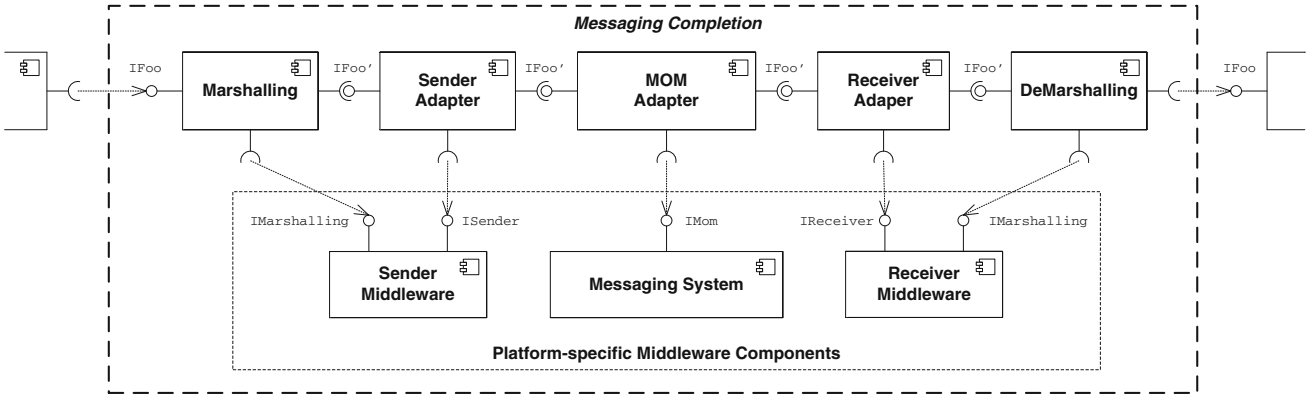



**Fig. 20** Components of the messaging completion

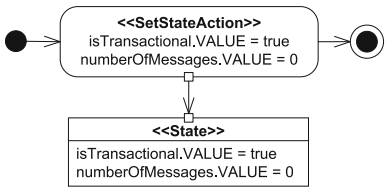



**Fig. 21** Starting a new transaction

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`: To start a transaction, the sender has to explicitly call the `startTransaction` method. Its behaviour (see Fig. 21) 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` is called, all the messages sent in the following become part of a new transaction until `commitTransaction` is executed. The behaviour of the `MOM Adapter` varies for transactional and non-transactional messages (see Fig. 22).

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 queuing 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.

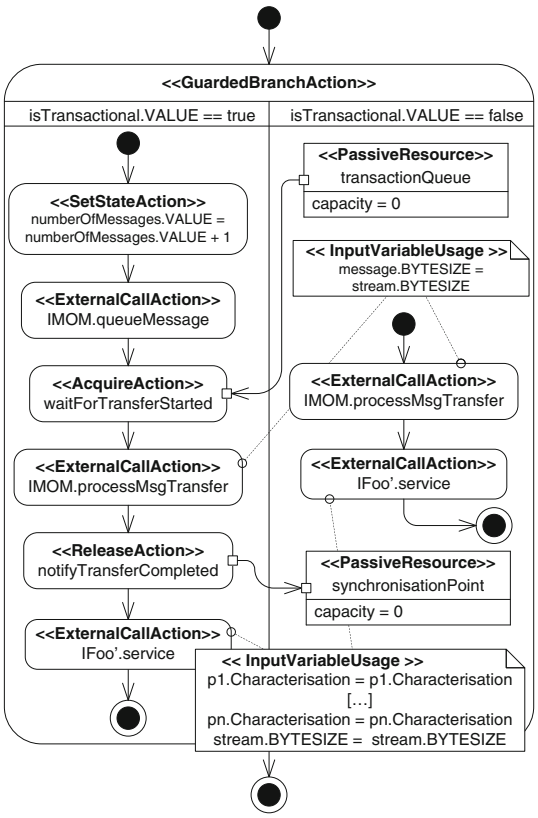



**Fig. 22** MOM adapter: message transfer

When a transaction is committed and the messages blocked at the `transactionQueue` are released, the `MOM Adapter` processes the message transfer (`IMOM.process MessageTransfer`). Furthermore, it notifies the behaviour of `commitTransaction` that the message has been transferred (`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 sent. Figure 23 shows the behaviour executed to commit a

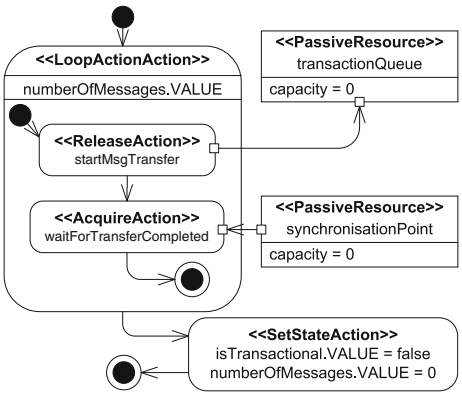



**Fig. 23** MOM adapter: commit transaction

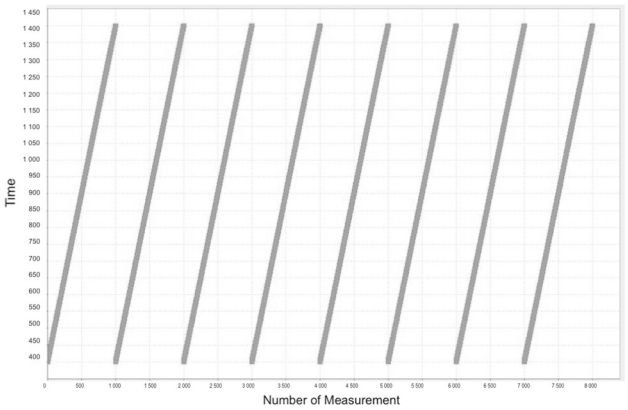



**Fig. 24** Predicted delivery times for messages in a transaction (one point stands for exactly one measurement in simulation of one message)

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

Prediction results for the stateful model of MOM: Figure 24 shows the prediction results for transactional messages using the models presented in this section. The corresponding real measurement is shown in Fig. 18. The predictions correctly reflect the dependence of a message's delivery time on its position in the transaction. Furthermore, the predicted delivery times range from 400 to 1,400 ms, corresponding to the observed delivery times. Moreover, the introduced model allows in-depth analysis of the performance impact resulting from the usage of transactions: the suitable size of the transaction can be optimised using such model, performance problems related to transactions can be easily identified and periodical utilisation of resources can be observed.

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 by less than 4 %. These results indicate that the extension of the model based on *PCM Stateful* can accurately predict the influence of (successfully completed) transactions on the delivery time of a message. Even more important as the prediction accuracy is that, in the extended model, we can observe periodical bust effects typical for state-dependent systems, which could not be observed in the previous model.

**Table 3** Measurement/prediction comparison

| Transaction size | Measurement (median, ms) | Prediction (median, ms) |
|---|---|---|
| 1 | 1.665917 | – |
| 2 | 2.506566 | 2.609999 |
| 4 | 4.157104 | 4.619999 |
| 10 | 9.145 595 | 9.050000 |
| 20 | 17.012373 | 17.079999 |
| 100 | 82.752583 | 85.440000 |
| 400 | 356.843626 | 360.980000 |
| 1,000 | 943.539863 | 943.370000 |

In the following, we shortly discuss the observations and the prediction results for the other two state-dependent representatives. As the used models are very similar to the previously discussed model extension for MOM, we do not describe these models for the remaining representatives in detail, instead, we focus on the observations resulting from such stateful models.

### 6.3 Configuration state: CACHE utilisation

The processes in real applications must be cache-efficient to utilise database effectively. However, the observable increase in performance is dependent on the current state of cache and its configuration. We evaluated the importance of cache configuration and how it impacts the performance of the system. The dependency on the current state of cache was represented by a configuration state, which was used to model the cache usage for all components deployed in one deployment container.

In our scenario, we used cache to support persisting of messages into the database on the headquarters server. In the

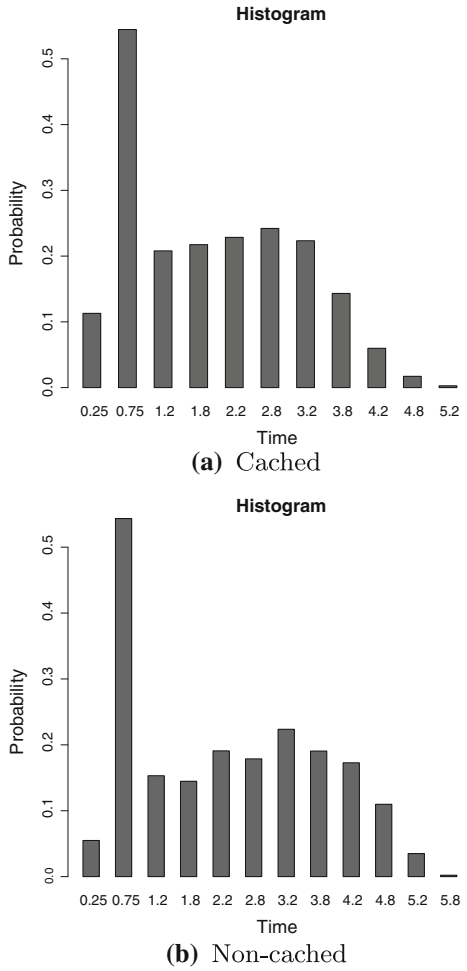



**Fig. 25** Comparison of the response time distribution for cached/non-cached requests

model, we considered limited capacity of cache and diverse sizes of messages. We observed increase of performance if cache was accessible and the transaction could be accessed in cache instead of the database. On the other hand, if the capacity of the cache was reached or had to be cleaned, we observed performance decrease. As the cache load resulted from a number of different sources and accumulated on the cache interface, we were able to observe this accumulated traffic and the increase/decrease in performance depending on how well the cache was balanced.

Our observation was that effective data caching (what data and when) is a balancing act as any data cached consumes resources and thus negatively impacts performance. However, if the cached data are often accessed by the clients, the decrease in performance is outweighted by the reduction of total number of database calls.

Our model was limited by the possibility to model memory and database calls by the underlying performance model. Therefore, our model misses the overhead resulting from database calls. However, even with this limitation, we were able to observe that our cache design for our usage scenario would not be of an advantage. In the predictions, we observed the difference in response time deviation between the model using cache and model without cache. Usage of cache increased the response time deviation from 0.9 to 1.2 ns. The mean values of both scenarios, however, were very similar: for the cached model 1.9 ns and non-cached model 1.8 ns. The difference between the models is better illustrated in Fig. 25. The observed effect can be explained through the design of our scenario that includes small queries on sets of different data. In such situation, cache does not deliver awaited performance increase. However, without considering cache in our model, we would not be able to reason about suitable cache usage and configuration at such early development stage. We demonstrate how the overall performance of a workload is dependent on cache usage and how cache modelling allows more detailed insights into the applications performance. Although, to balance cache usage properly, the stateful model needs to be accompanied with a detailed model of resources, such as memory and database.

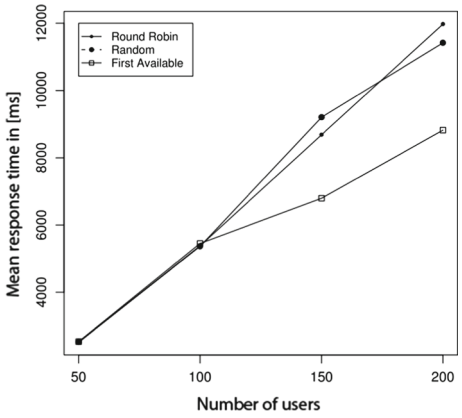
### 6.4 Persistent state: random vs. round robin load balancing

We further evaluated different strategies of request handling in the supply chain scenario. The supermarkets send their requests to and also get their responses returned from the headquarters server. However, considering replication on the side of headquarters server, we can provide shorter response times for the clients by distributing the incoming requests among available replicas.

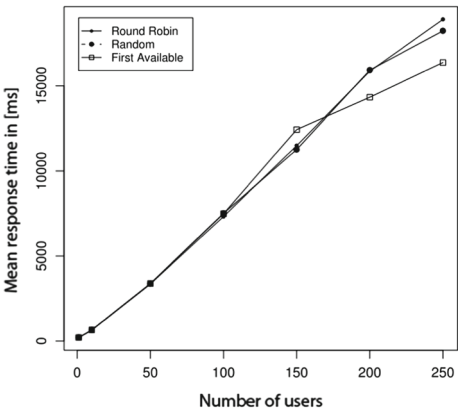
The replica chosen to process a request can be selected for each request or for all requests per user. Different strategies for the load balancing of requests are available, namely "Random" (i.e. chooses a replica randomly for each received request), "Round Robin" (i.e. following an order, every request is forwarded to the currently assigned replica, and the next replica is selected for the next user) and "First Available" (i.e. assigns each user to a randomly chosen replica). In the case of the last two strategies, the system needs to keep track of the current assignment of its replicas. All requests received from an assigned user are then always processed by the same replica. By these strategies, the pair is created once when a new user is registered and is used for the whole existence of the user in the system consistently for each request from this client.

Figure 26 summarises results of measurements on a real implementation with different load balancing strategies. We observed no large difference in the mean response times between these strategies. However, there are differences in the variance. To further investigate that behaviour, we conducted a series of simulation experiments.

The experiments demonstrated that, being dependent on the number of users, the difference between the strategies is




**(a)** Two replicas




**(b)** Three replicas

**Fig. 26** Comparison of the mean response times for requests, showing the three load balancing strategies

varying. We conducted simulation experiments with PCM for larger number of users. The experiments show that the user-specific persistent state holding information about mapping between users and replicas in the "First Available" load balancing strategy influences performance. First, the response time for models using "First Available" strategy varies less than the response time using the "Random" load balancing. Second, with the higher number of users, both strategies start to converge to the similar mean value and the difference between the two strategies is for the higher load smaller. Thus, there exist intervals based on number of users, where the accuracy of prediction with stateless model (e.g. "Random") is very high (especially by highly utilised systems). However, in the case of smaller load, the prediction error grows, for example, the predicted response times for 10 (or 1,000) users range from 10 to 95 ms (or 2,500–4,400 ms) with the "Random" strategy, which corresponds to the response times in the range from 22 to 33 ms (or 3,300–3,400 ms) for the "First Available" load balancing strategy. We observed that the variance for both strategies differs significantly, while the mean value is comparable. In Fig. 27,

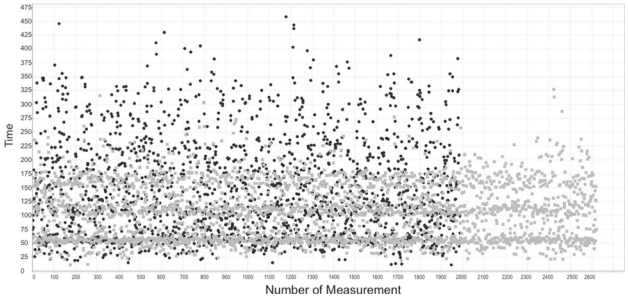



**Fig. 27** Comparison of the response times using "Round Robin" (*light*) and "Random" (*dark*) load balancing strategy (one point stands for exactly one measurement)

we show comparison between the "Random" and "Round Robin" load balancing strategies that illustrate the previously discussed observations. "Random"-stateless model of load balancing demonstrates higher variability than can be observed by "Round Robin"-stateful model, however, the mean values are very similar. Moreover, the periodical utilisation of resources can be observed in the case of "Round Robin".

This experiment shows that, when having different usage scenarios for the system, we have to test each usage scenario to decide about suitable load balancing strategy. In this case, pure stateless model has a problem with the accuracy of prediction and certain effects cannot be observed using such model. However, even if the mean and median values correlate for all the strategies to the similar value, the distributions of probabilities show major inaccuracy. Using the stateful model, it is possible to predict the systems behaviour resulting from selected load balancing strategy with higher accuracy. Although, we do not aim to contrast different load balancing techniques using presented model, the differences between them could be easily observed.

## 7 Discussion

The decision about an appropriate abstraction of state modelling in component-based software systems is a very complex task. As we show in Sect. 5, there are many aspects that influence the decision significantly. In the experiments, we have identified a number of situations when the probabilistic abstraction introduces high-prediction inaccuracies, even if the transition probabilities are estimated as precisely as possible (see Fig. 16 for instance). At the same time, the expected increase in model size may anyway discourage software engineers from including the stateful information into their models.

Although we have identified a number of aspects that indicate a low model-size increase in some situations, it is still very likely that stateful models have very high 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). On the other hand, we have observed that the results of stateful analysis tend to have much smaller variance, which also influences the time necessary to execute a simulation run. The higher variability of stateless models 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 also on the length of the simulated trace. However, explicitly modelled states have only little effect on the length of simulation traces, which mainly depends on the modelled software architecture (e.g. loops dependent on a state value). At the same time, even if the stateful model is significantly larger, 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.

When studying the performance impact of state modelling in Sect. 5, we have compared stateful models to their approximations with probabilistic models. As shown in the text, even if the probabilities in the stateless models reflect system usage and environment, the results of the performance evaluation may deviate significantly from the stateful models. The deviation is best visible on the probability distribution of the response-time values and the time series, which are the most fine-grained response-time metrics. Also the variance and best/worse case are very different, with a higher variance of stateless models. On the other hand, the median and mean values used to be quite stable, deviating often only slightly from the stateful model.

There are many types of systems, where the probabilistic models can approximate the stateful models very closely. For example, the influence of transactions (described in Sect. 6) 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 not only to know in advance the number of messages in a transaction (which is static and cannot change at runtime) but also the influence of a message on the transaction's delay (which needs to be adapted for each change in the transaction size to get accurate predictions). Modelling transactional messages probabilistically result in a comparable distribution of response times. However, the model does not reflect the stochastic dependence 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 a state with a probabilistic abstraction results in a decreased possibility of reuse of the component's prediction model because the probabilities are specific for one system, one allocation and one usage profile.

## 8 Conclusion

The paper addresses the challenge of performance prediction for stateful component-based software systems. To achieve this aim, we have identified and localised possible types of state-related information in component-based software systems, and structured them into a classification scheme along two dimensions, the scope and place within a component-based software architecture. We have surveyed the capability of existing performance-prediction approaches to model the identified categories, and extended the *Palladio Component Model (PCM)* with necessary state-modelling constructs. Later, we identified the similarities and differences of the individual state categories with respect to their performance impact and model-size increase, and introduced and evaluated a number of heuristics summarising the advices to software engineers, helping them to competently decide on the appropriate abstraction of state modelling.

In future, we aim to study the possibility to develop automated evaluation methods deciding on the validity of the individual heuristics and updating the performance model accordingly. The first step is the decomposition of the heuristics to more specific characteristics that define exact conditions to be evaluated on the analysed model. The automatisation would also include the employment of expert techniques to determine an appropriate abstraction on the values of individual state-related model parameters, to keep the model size and model accuracy balanced. Another aim of our ongoing research is to examine the impact of the hardware-specific state categories, which may reflect the availability and speed (based on the actual workload) of system hardware resources.

## References

1. Bagrodia, R., Shen, C.: Midas: integrated design and simulation of distributed systems. Trans. Softw. Eng. **17**(10), 1042–1058 (1991)
2. Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development: a survey. IEEE Trans. Softw. Eng. **30**(5), 295–310 (2004)
3. Becker, S., Grunske, L., Mirandola, R., Overhage, S.: Performance prediction of component-based systems. In: Architecting Systems with Trustworthy Components. LNCS, vol. 3938, pp. 169–192. Springer, Berlin (2006)
4. Becker, S., Happe, J., Koziolek, H.: Putting components into context: supporting QoS-predictions with an explicit context

model. In: Proceedings of the Workshop on Component Oriented Programming (WCOP) (2006)
5. Becker, S., Koziolek, H., Reussner, R.: The palladio component model for model-driven performance prediction. J. Syst. Softw. **82**(1), 3–22 (2009)
6. Bertolino, A., Mirandola, R.: Modeling and analysis of non-functional properties in component-based systems. In: Proceedings of the International Workshop on Test and Analysis of Component-Based Systems (TACoS). ENTCS, vol. 82, pp. 158–168. Elsevier, Amsterdam (2003)
7. Bertolino, A., Mirandola, R.: CB-SPE tool: Putting component-based performance engineering into practice. In: Proceedings of Component-Based Software Engineering (CBSE). LNCS, vol. 3054, pp. 233–248. Springer, Berlin (2004)
8. Diaconescu, A., Murphy, J.: Automating the performance management of component-based enterprise systems through the use of redundancy. In: Proceedings of Conference on Automated software engineering (ASE). IEEE (2005)
9. Firus, V., Becker, S., Happe, J.: Parametric performance contracts for QML-specified software components. In: Proceedings of Formal Foundations of Embedded Software and Component-based Software Architectures (FESCA). ENTCS, vol. 141, pp. 73–90. Elsevier, Amsterdam (2005)
10. Gallotti, S., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Quality prediction of service compositions through probabilistic model checking. In: Proceedings of Quality of Software Architectures (QoSA). LNCS, vol. 5281, pp. 119–134. Springer, Berlin (2008)
11. Hamlet, D.: Subdomain testing of units and systems with state. In: Proceedings of Symposium on Software testing and analysis (ISSTA). ACM (2006)
12. Hapner, M., Burridge, R., Sharma, R., Fialli, J., Stout, K.: Java Message Service Specification-Version 1.1. http://www.oracle.com/technetwork/java/jms/index.html, (2012)
13. Happe, J., Becker, S., Rathfelder, C., Friedrich, H., Reussner, R.H.: Parametric performance completions for model-driven performance prediction. Perform. Eval. **67**(8), 694–716 (2009)
14. Happe, J., Westermann, D., Sachs, K., Kapova, L.: Statistical inference of software performance models for parametric performance completions. In: Research into Practice-Reality and Gaps (Proceedings of QoSA 2010). LNCS, vol. 6093, pp. 20–35. Springer, Berllin (2010)
15. Hissam, S., Moreno, G., Stafford, J., Wallnau, K.: Enabling predictable assembly. J. Syst. Softw. **65**(3), 185–198 (2003)
16. Kapova, L., Buhnova, B., Martens, A., Happe, J., Reussner, R.: State dependence in performance evaluation of component-based software systems. In: Proceedings of the Joint WOSP/SIPEW International Conference on Performance engineering (ICPE). ACM (2010)
17. Kapova, L., Buhnova, B., Reussner, R.: Stateful software performance egineering. http://sdqweb.ipd.kit.edu/wiki/Stateful_Software_Performance_Engineer (2012)
18. Kapova, L., Reussner, R.: Application of advanced model-driven techniques in performance engineering. In: Computer Performance Engineering. LNCS, vol. 6342, pp. 17–36. Springer, Berlin (2010)
19. Koziolek, H.: Performance evaluation for component-based software systems: a survey. Perform. Eval. **67**(8), 634–658 (2010)
20. Koziolek, H., Becker, S.: Transforming operational profiles of software components for quality of service predictions. In: Proceedings of Workshop on Component Oriented Programming (WCOP) (2005)
21. Koziolek, H., Becker, S., Happe, J.: Predicting the performance of component-based software architectures with different usage profiles. In: Proceedings of Quality of Software Architectures (QoSA). LNCS, vol. 4880, pp. 145–163. Springer, Berlin (2007)
22. Koziolek, H., Happe, J., Becker, S.: Parameter dependent performance specifications of software components. In: Proceedings of Quality of Software Architectures (QoSA). LNCS, vol. 4214, pp. 163–179. Springer, Berlin (2006)
23. Liu, Y., Fekete, A., Gorton, I.: Design-level performance prediction of component-based applications. Trans. Softw. Eng. **31**(11), 928–941 (2005)
24. Meyerhofer, M., Meyer-Wegener, K.: Estimating non-functional properties of component-based software based on resource consumption. In: Proceedings of the Software Composition Workshop (SC). ENTCS, vol. 114, pp. 25–45. Elsevier, Amsterdam (2005)
25. Microsoft Corporation: COM: Component Object Model Technologies. http://www.microsoft.com/com/, (2012)
26. Mos, A., Murphy, J.: Performance management in component-oriented systems using a model driven architecture approach. In: Proceedings of Enterprise Distributed Object Computing Conference, pp. 227–237. IEEE (2002)
27. Object Management Group: CORBA Component Model 4.0 Specification. Tech. Rep. formal/06-04-01, Object Management Group (2006)
28. Object Management Group: MOF 2.0 Query/View/Transformation, version 1.0 (2008)
29. Sentilles, S., Vulgarakis, A., Bureš, T., Carlson, J., Crnković, I.: A component model for control-intensive distributed embedded systems. In: International Symposium on Component-Based Software Engineering (CBSE). LNCS, vol. 5282, pp. 310–317. Springer, Berlin (2008)
30. Smith, C.U., Williams, L.G.: New software performance antipatterns: more ways to shoot yourself in the foot. In: Proceedings of International CMG Conference. Computer Measurement Group (2002)
31. Standard Performance Evaluation Corp.: SPECjms2007 Benchmark. http://www.spec.org/jms2007/, (2007)
32. Sun Microsystems: Java System Message Queue -Version 4.3. http://java.net/downloads/mq/OpenMQ4.3Related/MQ4.3_RelNotes_preFCS.pdf (2012)
33. Sun Microsystems: Enterprise JavaBeans 3.0 Specification. http://java.sun.com/ (2006)
34. Szyperski, C.: Component software—beyond object-oriented programming. Addison-Wesley, Reading (2002)
35. Westermann, D., Happe, J., Hauck, M., Heupel, C.: The performance cockpit approach: a framework for systematic performance evaluations. In: Proceedings of the EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), pp. 31–38. IEEE Computer Society (2010)
36. Woodside, M., Franks, G., Petriu, D.C.: The future of software performance engineering. In: Proceedings of Conference on Software Engineering (ICSE), pp. 171–187. IEEE (2007)
37. Zimmerova, B.: Modelling and formal analysis of component-based systems in view of component interaction. Ph.D. thesis, Masaryk University, Czech Republic (2008)
38. Zimmerova, B., Vařeková, P., Beneš, N., Černá, I., Brim, L., Sochor, J.: The common component modeling example: comparing software component models. LNCS, vol. 5153, chap. Component-Interaction Automata Approach (CoIn), pp. 146–176. Springer, Berlin (2008)

## Author Biographies

**Lucia Happe** is a senior researbreakcher at the Karlsruhe Institute of Technology (KIT) in the group of Software Design and Quality. She received her Ph.D. in computer science from the KIT as well. In 2008, she was awarded a Ph.D. scholarship from the Deutscher Akademischer Austauschdienst (DAAD). She got her diploma in computer science from the University of Kosice in Slovakia. Her interests include software architectures, model-driven development methods, and model-based quality prediction.

**Barbora Buhnova** is an assistant professor of software engineering at Masaryk University, Czech Republic. She received her Ph.D. degree in Computer Science from the same university, for application of formal methods in component-based software engineering. She continued with related topics as a postdoc researcher at University of Karlsruhe, Germany, and Swinburne University of Technology, Australia. Bara's research interests are centred around quantitative analysis of software architectures.

**Ralf Reussner** is full professor of software engineering at the Karlsruhe Institute of Technology (KIT) since 2006 and scientific executive of the IT Research Centre in Karlsruhe (FZI). He graduated in Karlsruhe with a Ph.D. in 2001 and was with the DSTC Pty Ltd.,break Melbourne, Australia, afterwards. From 2003 till 2006, he held a junior professorship at the University of Oldenburg, Germany. Ralf's research group works in the area of component-based software design, software architecture and predictable software quality.