Model-Driven Performance Engineering of Self-Adaptive Systems: A Survey

Matthias Becker  
Software Engineering Group,  
Heinz Nixdorf Institute,  
University of Paderborn  
Zukunftsmeile 1  
33102 Paderborn, Germany  
matthias.becker@upb.de

Markus Luckey  
Department of Computer Science, University of Paderborn  
Zukunftsmeile 1  
33102 Paderborn, Germany  
markus.luckey@upb.de

Steffen Becker  
Software Engineering Group,  
Heinz Nixdorf Institute,  
University of Paderborn  
Zukunftsmeile 1  
33102 Paderborn, Germany  
steffen.becker@upb.de

ABSTRACT

To meet quality-of-service requirements in changing environments, modern software systems adapt themselves. The structure, and correspondingly the behavior, of these systems undergoes continuous change. Model-driven performance engineering, however, assumes static system structures, behavior, and deployment. Hence, self-adaptive systems pose new challenges to model-driven performance engineering. There are a few surveys on self-adaptive systems, performance engineering, and the combination of both in the literature. In contrast to existing work, here we focus on model-driven performance analysis approaches. Based on a systematic literature review, we present a classification, identify open issues, and outline further research.

Categories and Subject Descriptors
D.2.8 [Software Engineering]: Metrics—Performance measures; D.2.11 [Software Engineering]: Software Architectures—Languages

Keywords
software performance, model-driven performance engineering, self-adaptation, self-*

1. INTRODUCTION

Modern business information systems run in highly dynamic environments. Dynamics range from unpredictably changing numbers of concurrent users asking for service to virtualized infrastructure environment with unknown load caused in neighboring virtual machines or varying response times of required external services. Despite these dynamics, these systems are expected to fulfill their performance requirements. Designers achieved this in the past by over-provisioning of hardware, which is, however, neither cost-effective nor energy-preserving. Self-adaptation is one primary means developed over the last years to cope with these challenges. The idea is that systems always react to their dynamic environment by restructuring their components and connectors, exchanging components or services, or altering their hardware infrastructure.

To deal with performance requirements of classical, non-adaptive systems, researches have developed model-based and model-driven performance engineering approaches [5]. These approaches allow early design-time performance evaluations based on system models to validate performance requirements or answer hardware sizing questions. However, classical performance engineering approaches use non-adaptive system models (e.g., plain UML2 models) which do not support an explicit adaptation viewpoint. Consequently, as this viewpoint models the self-adaptive behavior of today’s systems, we cannot apply classical performance engineering approaches without extensions. This raises the question how and to which extent newly developed approaches for model-driven performance engineering of self-adaptive system take this novel viewpoint into account.

There are already some literature surveys on modeling self-adaptive systems, classical performance engineering, and even a few on performance engineering in self-adaptive environments partially answering our question. However, we focus on model-driven approaches as we aim for full automation and design-time concerns, e.g., will the system’s adaptation ruleset ensure stability.

The contribution of this paper is an initial literature review of the current state-of-the-art in performance engineering for self-adaptive systems. In our survey, we restrict our focus to model-driven approaches with explicitly modeled adaptation rules in the business information systems domain. We classify the identified approaches and point out future research to fill open gaps.

In the following section, we give a detailed motivating example from which we derive our research questions. To answer them, Section 3 briefly revisits background knowledge on self-adaptive system modeling and performance engineering. We present our survey’s classification in Section 4.1. We then use it to evaluate the papers in our study (Section 4.2). Finally, we discuss our findings in Section 4.3. In Section 5, we discuss related literature surveys before we conclude in Section 6.
The purpose of our survey is to identify and understand model-driven performance engineering approaches for self-adaptive business information systems from the point of view of software engineers and researchers. In this section, we outline our motivation by providing a small example scenario. Using the presented scenario, we illustrate some challenges in model-driven performance engineering of self-adaptive systems and formulate research questions.

Consider a web-based application implemented as a three-tier architecture, as illustrated in Figure 1. To achieve a reasonable performance and efficiency of the system, various factors have to be considered, such as the changing system load and load peaks.

Classical model-driven performance engineering can help to determine the best sizing. Therefore, we model our system architecture with performance annotations, as shown in Figure 1, and derive an analysis model, e.g., a Layered Queuing Network (LQN), via automatic model transformations. Solving the analysis model gives us a prediction of how our system will perform. However, due to the specification of peak work loads in the model, our system will still be overdimensioned for the average work load. The average utilization of a real world server system, similar to our example, is estimated to be about 5% to 20% [1], resulting in unnecessary costs.

Infrastructure-as-a-Service (IaaS) cloud environments are advocated to solve this problem and to save costs. In IaaS clouds, resources, i.e., CPU, HDD, and RAM, can be allocated and released on demand during run-time. Running our system in an IaaS cloud allows us to allocate only as many resources as needed to handle the actual load.

We can design and implement a self-adaptive system which smartly adapts to its load to benefit from the flexibility an IaaS cloud offers. In Figure 1, an adaptation rule is defined using the Adapt Case language [17]. The rule periodically checks our system’s load and instantiates additional storage servers whenever the load is higher than 80%. The challenge is to design the right adaptation strategies and their parameters for our system, e.g., we need to decide which, when, and how many resources have to be allocated or released. Additionally, we want our adaptation strategies to be cost-efficient, i.e., allocating as little resources as possible at any time.

Classical performance engineering cannot be applied for self-adaptive systems since static system structures are assumed. However, there exist some approaches which apply model-driven performance engineering on self-adaptive systems. We want to identify, understand, and examine these approaches according to their capabilities and limitations.

For this purpose, we formulate two research questions, R1 and R2, which we want to address in this survey: (R1) What can be achieved with the state-of-the-art model-driven performance engineering approaches for self-adaptive systems? What are the limitations? (R2) What are the missing aspects raising a need for further research?

Model-driven performance engineering of self-adaptive systems is a combination of three research areas, as illustrated in Figure 2: Self-adaptive system analysis, performance engineering, and model-driven engineering. In order to answer our research questions, we need to understand all three research areas and their characteristics. Hence, we conducted three steps: (S1) We identified some prominent classes of modeling and analysis approaches for self-adaptive systems, (S2) we identified some prominent classes of model-driven performance engineering approaches, and (S3) we surveyed approaches which apply model-driven performance engineering to self-adaptive systems, which we present in Section 4.

### 3.1 Engineering of Self-Adaptive Systems

Existing and emerging approaches for modeling and analyzing self-adaptive systems most often concentrate on the functional correctness, e.g., stability. Performance analysis is hardly addressed. Focusing on business-information systems and high-abstraction modeling languages that can be used in model-driven engineering approaches, there are few prominent representatives for model-based design approaches for the analysis of self-adaptive systems.

**Goal-oriented requirements engineering** approaches use goal trees to model requirements for self-adaptive systems [20]. Controlled natural language (RELAX) combined with goal-oriented requirements engineering is presented. RELAX can be translated into temporal logic and thus be analyzed. Adapt Cases are a UML use case based modeling language to explicitly describe adaptivity on a platform-independent modeling level [17]. The level of detail is sufficiently high to perform first quality checks as shown in [16]. Basically, functional quality properties are checked, e.g., stability, deadlock-freedom, and loop-freedom. Strands is one of several concepts that have been proposed as a UML profile by Hebig et al [10]. The profile is meant to model control loops as first class entities when architecting self-adaptive systems with UML. In [15] the authors show how to design self-adaptive systems using the architecture description language Darwin and a Three-Layer Architecture. The authors use a model...
checking approach to identify valid configurations. As a representative for domain-specific languages of self-adaptive systems, in [7] Fleurey et al. describe an approach to specify self-adaptive systems in terms of variants using an EMF-based meta-model that is instantiated using tabular-like editors. Further, they use the constraint solver Alloy to check functional properties and perform simulations.

The commonality of all these approaches is their focus on functional analysis and accordingly their lack of modeled information for performance analyses.

3.2 Performance Engineering

In opposite to classical, model-based performance engineering, architectural system models are a first level entity in model-driven performance engineering. These system models are transformed to analysis models by means of model-transformations and subsequently analyzed. Ideally, performance problems are then propagated back to the architectural model. In contrast to model-based approaches [2, 5] in model-driven approaches the analysis models are automatically derived from architectural models. We further distinguish here two major classes in model-driven performance engineering: component-based and bridge model-based.

Component-based performance engineering is intended to determine the performance of a software system which is assembled from components. A representative for model-driven component-based performance engineering is Palladio [3]. Palladio is an Eclipse-based tool suite for designing and analyzing component-based software. The Palladio Component Model (PCM) uses a UML-like notation extended with performance annotations.

Bridge model-based approaches especially aim to alleviate the transformation by introducing a bridge model between architectural models and analysis models. This makes it necessary to implement two transformations, but enables a flexible reuse of once defined transformations. The KLAPER approach [9] by Grassi et al. is a prominent representative of bridge model approaches. The approach is not meant to provide own architectural models, but transformations from the bridge model to analysis models. However, none of the existing approaches considers self-adaptive system structures.

4. SURVEY

We outline the review method and classification scheme (4.1) which we used to survey approaches for performance engineering of self-adaptive systems (4.2) and discuss the results and give recommendations for further research (4.3).

In order to achieve objective and unbiased results, we conducted our review according to guidelines for systematic literature review [12] by Kitchenham and Charters. Our data sources for the survey have been common scientific search engines\(^1\). We conducted our search during the period from October 2011 to February 2012. We included approaches found with combinations from both of the following keyword groups: \{self-adaptation, self-*\} and \{performance engineering, performance analysis, QoS analysis\}. Since we are only interested in model-driven performance engineering of self-adaptive business information systems, we explicitly excluded approaches in the area of embedded system engineering as well as all non model-driven approaches. Model-driven approaches in our sense only include approaches using higher abstraction-level models to derive lower abstraction-level models via model transformations as defined by Stahl et al. [22].

4.1 Classification

Figure 3 shows a feature diagram of our classification scheme, which we will detail in the following. First, we started with a coarse-grained classification (top-level white features) based on our expertise. Second, we derived further classification criteria (white features) from the characteristics of self-adaptive system design approaches and performance engineering approaches, we presented in Section 3. Finally, we refined the classification based on our findings (grey features). We classify the approaches we have found according to four categories: adaptation, architecture, performance analysis, and applicability.

With the first classification criterion, adaptation, we classify the approaches according to which self-adaptation patterns and self-adaptation strategies are implemented. The most common self-adaptation patterns are feedback loops and layered approaches. The MAPE-K [19] is a feedback loop pattern, that divides the process of adaptation into four phases: monitor (M), analyze (A), plan (P), and execute (E). Data that is collected and used during adaptation is stored in the so-called knowledge base (K). The object being adapted is named managed element. Another often used self-adaptation pattern is the Three-Layer Architecture [15]. The bottom component control layer contains the interconnected components of the system with basic self-tuning capabilities. The change management layer effects changes to the underlying component architecture in response to new states (reported from below) or new objectives (reported from above). The uppermost goal management layer deals with change management plans in response to requests from the lower layer or from new goals. We further distinguish between two different adaptation strategies: reactive and proactive. We call an adaptation strategy reactive, if the system triggers its self-adaptation when a goal is already violated. If the system predicts that it might miss a goal some time in the near future and hence adapts itself preventively, we call that proactive.

The second classification criterion is the system architecture. The approaches we have found were restricted to either component-based architectures or service-oriented architectures (SOA). We also examine which modeling languages are used to model the system architecture, e.g., the UML or custom modeling languages.

Performance analysis is the third classification criterion where we classify the approaches according to when the approach is applied (time), which analysis method and models are used, and if transformations are provided. We distinguish between two different points in time when performance engineering can be applied in self-adaptive systems: at design-time and at run-time. Self-adaptive systems can adapt themselves to meet predefined goals. The goals may be functional goals, e.g., correctness of responses, or non-functional goals, like response time lower than 5ms, reliability over 97%, etc. A main challenge at design-time is to identify proper adaptation strategies, i.e., it has to be analyzed whether the rules are sufficient to achieve the system’s QoS goals assuming a certain system context. At run-time performance engineering can be applied to measure the real

\(^1\)Google Scholar, Microsoft Academic Search, DBLP
system’s context and to predict performance trends. Furthermore, there are two types of analysis methods: analytical and simulation. Different analysis models can be used, e.g., LQNs, Petri nets, or Markov models. Whether automatic transformations from architectural models to analysis models are provided is another classification criterion.

Finally, we examined the applicability of the approaches. On the one hand, the applicability depends on to which extend the approach provides tool support. That is whether only the performance analysis is tool-supported or a complete model-driven engineering from architecture modeling to performance analysis is supported. On the other hand, the approaches are validated by proof-of-concept implementations, or case studies.

4.2 Results

The approaches we present have been identified by our review method. We examined and classified these approaches according to the classification scheme presented in the previous section. Our presentation is divided into two sub-categories according to when the presented approaches are applied: at design-time or at run-time. Table 1 shows the results of our evaluation.

Design-time The D-KLAPER approach [8] by Grassi et al. focuses on a transformation chain from design models to analysis models. The approach uses a bridge model to alleviate the transformation from architectural models to analysis models. The bridge model is used to describe an adaptive system as a set of resources (hardware and software) which offer and require services. Adaptations are modeled as a special kind of service call. Each adaptation service call can be annotated with quality attributes, e.g., failure rate. There is no modeling support for system design models. UML SPT is used instead. Furthermore, there is neither support for modeling adaptation rules nor were transformations from input models to intermediate models presented yet. For the analysis, several analysis models are supported but may require additional transformations. To model the dynamics of self-adaptive systems, (Semi-) Markov Reward Models (SMRM) are chosen as analysis models. For the analysis it is assumed that adaptations do not happen at the same frequency as system events. The systems are considered to be in a steady state when analyzed. Adaptation costs are analyzed separately using the results from system analysis.

Meyer extends the PCM with reconfiguration strategies [18] in his Master’s thesis. An initial system configuration can be derived from a static Palladio Component Model. Using this static architectural system model and reconfiguration strategies, the self-adaptive system’s performance is evaluated by simulating a queuing network. The simulation uses a PCM model interpreter and updates the PCM model according to the reconfigurations during simulation. However, the simulation analysis is limited to the analysis of transient states yet. That is a complete set of adaptation strategies cannot be simulated. Due to the reuse of the Palladio platform this approach provides reasonably good tool support for creating architectural models and performance analysis. However, the creation of reconfiguration strategies using Story Diagrams is not integrated into the Palladio suite yet.

Run-time With Descartes [13], Kounev presents a vision for a proactive performance engineering approach in self-adaptive cloud systems. The approach is to integrate QoS models into the system’s components: in that way a component is self-aware of its own performance properties. Furthermore, the approach is planned to provide automatic model extraction based on performance monitoring data. The model extraction is used to generate analysis models on-the-fly. These models are used to predict the performance running system. The author also plans to predict the system’s workload and the effect of a self-adaptation a priori. Hence, the Descartes approach incorporates three control loops: A MAPE-K control loop for self-adaptation, a loop for refining and calibrating the online analysis models, and a control loop to forecast the workload.

QoSMOS [4] is an approach to model and implement self-adaptive systems whose self-adaptation is driven by QoS requirements. The system designer has to manually derive an analysis model from the architectural model. However, automatic model transformations are planned. The manually derived analysis model serves as initial input for the online performance analysis and is updated during run-time, using the KAMI approach. The QoSMOS approach integrates several tools into a complete tool suite to model the architecture and QoS requirements of a self-adaptive system.

Elkodary et al. present an approach, named FUSION [6], which uses feature diagrams as the system model. The system’s architecture variants are represented as a feature diagram, where a feature configuration reflects the current architecture configuration of the system. Hence, self-adaptation is realized by switching between different system configurations. The self-adaptation in FUSION is goal-driven, i.e., relying on predefined functional or non-functional goals. Influence of an adaptation is learned and stored in the system’s knowledge base. The authors have extended the XTEAM tool to support the modeling of goals and features. Since this approach is based on learning no analysis models or transformations are required.

The Self-Adaptive Framework for Concurrency Architec-
tures (SAFCA) [23] is a self-adaptation approach based on the MAPE-K control loop. The basic idea is that concurrent architectures for a system are compared regarding their performance during run-time and the architecture which promises better performance will be selected as the new system architecture. Performance values are continuously monitored and compared to historical performance values stored in a knowledge database. Self-adaptation will be triggered when the current performance values indicate a bad performance. For this purpose, the performance is analyzed using queuing networks which correspond to the concurrent architectures.

4.3 Discussion

Looking at the evaluation in Table 1, we make the following observations: Half of the surveyed approaches use reactive self-adaptation strategies. A proactive strategy, however, is preferable because it enables the operation of a system without violating its goals. Well-approved self-adaptation patterns, like feedback loops and the three-layer pattern, are not applied by any of the surveyed approaches.

The used architectural models vary in the used notation but most approaches assume component-based architectures. Queuing networks are used as analysis model in half of all approaches. D-KLAPER uses SMRM and FUSION is a learning-based approach. Model-transformations from architecture models to analysis models are rarely provided. D-KLAPER provides a bunch of transformations from its bridge models to analysis models, SimuLizar simulates an architecture model directly. Only the QoS-MOS approach provides complete model-driven tool support. That is from modelling architectural models, automatic derivation of analysis models, and analysis, and evaluation of the analysis. However, we could not find any tool support for evaluating alternative adaptation strategies.

All approaches have shown their applicability by providing a proof-of-concept implementation. However, none of the approaches have conducted a case study to validate their applicability. Hence, the applicability in practice is still questionable.

In general, we have noticed a discrepancy between analytical approaches for self-adaptive system engineering and performance engineering. In the area of self-adaptive system engineering the analytical methods are mainly used to evaluate whether adaptation rules maintain the system’s functionality. Non-functional aspects such as performance are not analyzed at all or just informally. In the area of performance engineering, dynamic contexts such as varying deployments of systems running in IaaS clouds are neglected. These dynamic contexts, however, are the original reason why a system has to adapt itself. Furthermore, no approach does support engineers in deciding on adequate adaptation strategies. This is due to the lack of analyzing the adaptation rules themselves. Instead, only steady states of the system are analyzed. Reconfiguration times and costs are uncommonly neglected.

We recommend implementing complete model-driven tool support including modeling of self-adaptive systems with explicit adaptation strategies. For convenience, tools should provide automatic transformations to code and to analysis models as well as comprehensive evaluation of analyses. For design-time analysis, additional models should be introduced such as dynamic usage models which take varying contexts and explicit reconfiguration rules into account. Finally, we recommend conducting case studies to prove the applicability of performance engineering of self-adaptive systems based on real implementations.

5. RELATED WORK

The research community has provided several surveys on both research areas we addressed in this survey: self-adaptive system modeling and performance engineering. Up to now, these research areas were isolated from each other but are converging recently.

Kephart identifies some of the main scientific and engineering challenges of autonomic computing [11]. His focus is mainly on architectures, technologies, and (formal) tools. Salehie et al. group existing approaches by the corresponding discipline [21]. Disciplines include software engineering, artificial intelligence, control theory, and distributed computing.

Koziólek surveys approaches for performance prediction of component-based software systems [14]. The survey examines 13 approaches concerning general features, modeling formalisms, maturity, and applicability in industry. Koziólek identified no component-based performance prediction approach addressing self-adaptive systems.

To summarize, none of these surveys investigates the combination of self-adaptive system modeling and performance engineering in particular. Therefore, we hope to provide the research community with a valuable new viewpoint on current research results and challenges.

6. CONCLUSION

In this paper, we identified and classified approaches for performance engineering of self-adaptive systems. We examined existing approaches according to their capabilities and limitations and revealed some discrepancies between self-adaptive system modeling and performance engineering. Our survey equally helps engineers and researchers. Engineers get an overview of existing approaches and researchers can tackle the limitations we have identified.

7. ACKNOWLEDGMENTS

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre “On-The-Fly Computing” (CRC 901).

8. REFERENCES

<table>
<thead>
<tr>
<th></th>
<th>D-KLAPER</th>
<th>SimuLizar</th>
<th>Descartes</th>
<th>FUSION</th>
<th>QoSMOS</th>
<th>SAFCA</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Adaptation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pro-/Reactive</td>
<td>proactive</td>
<td>reactive</td>
<td>proactive</td>
<td>reactive</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Pattern</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Architecture</strong></td>
<td>UML-based</td>
<td>PCM</td>
<td>(unspecified)</td>
<td>feature diagram</td>
<td>BPEL</td>
<td>(unspecified)</td>
</tr>
<tr>
<td><strong>Architectural Model</strong></td>
<td>components</td>
<td>components</td>
<td>(unspecified)</td>
<td>components</td>
<td>SOA</td>
<td>(unspecified)</td>
</tr>
<tr>
<td><strong>Performance Analysis</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Method</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Analysis Model</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transformation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Additional Models</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Tool Support</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Analysis</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complete MD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Validation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proof-of-Concept</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Case Study</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Surveyed approaches