

Performance-oriented Design Space Exploration

Anne Martens and Heiko Koziolok
Institute for Program Structures and Data Organisation
Faculty of Informatics
University of Karlsruhe (TH), Germany
Email: {martens | koziolok }@ipd.uka.de

Abstract—Architectural models of component-based software systems are evaluated for functional properties and/or extra-functional properties (e.g. by doing performance predictions). However, after getting the results of the evaluations and recognising that requirements are not met, most existing approaches leave the software architect alone with finding new alternatives to her current design (e.g. by changing the selection of components, the configuration of components and containers, the sizing). We propose a novel approach to automatically generate and assess performance-improving design alternatives for component-based software systems based on performance analyses of the software architecture. First, the design space spanned by different design options (e.g. available components, configuration options) is systematically explored using metaheuristic search techniques. Second, new architecture candidates are generated based on detecting anti-patterns in the initial architecture. Using this approach, the design of a high-quality component-based software system is eased for the software architect. First, she needs less manual effort to find good design alternatives. Second, good design alternatives can be uncovered that the software architect herself would have overlooked.

I. INTRODUCTION

Performance problems are continuously prevalent in many software systems [1]. Model-based prediction methods [2] try to tackle these problems during early design phases to avoid the problem of implementing architectures which are not able to fulfil certain performance goals.

However, after getting the results of the performance prediction and potentially recognising that requirements are not met, most existing approaches leave the software architect alone with finding new alternatives to her current design (e.g. by changing the selection of components, the configuration of components and containers, the sizing), when recognising that a design is sub-optimal and does not meet her requirements.

To meet the difficulty of solving performance problems, we propose a novel approach to automatically generate and assess design alternatives for component-based software systems based on performance analyses of the software architecture. To do so, we use a mixed approach of metaheuristic search techniques [3] and anti-pattern detection to systematically create and evaluate new architecture candidates. First, the design space spanned by different design options (e.g. available components configuration options) is systematically explored using metaheuristic search techniques such as random-restart hill climbing, genetic algorithms or others, using the performance analysis as the decision criterion. Second, the performance analysis results of the architecture is searched for

performance problems such as anti-patterns and bottlenecks, so that new candidates can be generated to solve these detected performance problems. Here, anti-patterns can be removed by applying the corresponding solution documented with the anti-pattern. Bottlenecks can also be met by finding a good allocation or changing the resource dimensioning. The results can be presented to the software architect and thus provides her with detailed feedback.

The approach is an iterative process with manual intervention by the software architect. For example, she can decide whether introducing cache is possible for a given communication between two remote components. Except for this step, the rest of the approach can be completely automated.

Our proposed approach can improve a component-based system's architecture for its performance. This has two benefits for the software architect: 1) The approach saves her time, as she does not have to explore the design space manually and 2) the approach might result in better architectures than a manual exploration, as many more candidates can be evaluated in a short time. To validate these benefits, we plan to compare the application of this approach with a conventional, manual approach in an empirical study.

The contribution of this paper is the proposition of an design space exploration process to a) provide a software architect with detailed feedback on the performance evaluation of her architecture and even further b) (semi-)automatically find better architecture candidate and thus improve the software architect's design.

This paper is organised as follows. Section II introduces the foundations of component-based software performance engineering, especially the Palladio Component Model (PCM) as one exemplary approach, and defines terms for the rest of the paper. Section III introduces the running example used in this paper to describe the design space exploration. Section IV explains the actual design space exploration process and embeds it into the process model for component-based software development proposed by [4]. Section V presents related work and section VI discusses open issues for further research. Finally, section VII concludes.

II. COMPONENT-BASED SOFTWARE PERFORMANCE ENGINEERING

The following briefly explains the process model of CBSPE, describes the PCM for CBSPE, and defines several terms, which will be used in the remainder of this paper.

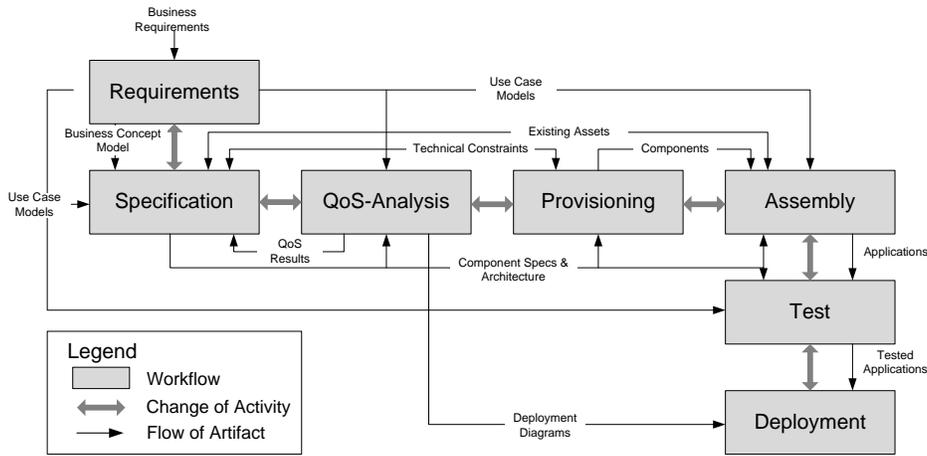


Fig. 1. CB-SPE Process Model

A. Process Model

Fig. 1 contains a development process model for component-based systems, which includes CBSPE [5]. This process model is an extended version of the CBSE-process model from Cheeseman and Daniels [4]. It covers the technical development part and neglects concurrent management tasks, such as scheduling and controlling tasks. The process model contains workflows (boxes), broad arrows indicating a change of activity, and slim arrows indicating an artefact flow. The workflows do not need to be traversed linearly, as backward steps into former workflows are possible.

Given business requirements in textual form, the process starts with the *requirements* workflow, where the informal business requirements are formalised. In the following *specification* workflow, the software architect designs the system's architecture using a model describing components and connectors. Existing component specifications (e.g., from repositories) can be used to build the model. Furthermore, the software architect can specify new components for special requirements. For CBSPE, the component specifications include execution times parameterised for input value and the underlying resources [6]. The specification workflow should result in an architectural design model capable of fulfilling the system's functional requirements.

The *QoS-analysis* workflow is then responsible for ensuring fulfilment of the system's extra-functional requirements. Therefore, the architectural model is augmented with information about the underlying resources (e.g., CPU speed, number of services, etc.) and system usage (e.g., the number of users, input parameters). For performance prediction, model-transformations convert this augmented model into a classical performance model, such as a queuing network, stochastic process algebra, or stochastic Petri-net. There are a number of analytical techniques and simulators for these models [7], which deliver the expected response times, throughputs, and resource utilisations.

If the performance predictions indicate that the performance requirements of the system cannot be met, the software

architect has to rework the architecture model (going back to the specification workflow) or renegotiate the requirements. The potential for altering the architecture model to improve performance properties is large. Many factors influence the performance of a component-based software architecture (e.g., thread pool sizes, operating system schedulers, communication protocols, etc.), but nowadays the software architect has to assess these influences manually and gets no support of possible improvements of the architecture.

If the performance predictions indicate that the performance requirements are fulfilled, the components of the system are build or bought from third-party vendors during the *provisioning* workflow. After provisioning, the software architect connects the components together (*assembly* workflow), so that the system is ready for (*testing* workflow) under controlled conditions. Finally, after successful testing, the system can be deployed at the customers site.

B. Palladio Component Model

The Palladio Component Model (PCM) is a modelling language specifically designed for CBSPE [6]. The language lets component developer specify the performance properties of component in a parametrised form and lets software architects combine these specifications to architectural models. Model transformations from the PCM target several performance models, such as stochastic regular expressions [8], layered queuing networks [9], and a discrete-event simulation [6].

PCM models can be configured according to a wide range of options. Configuration options are expressed as feature models. They allow for example configuring performance-relevant parameters of communication protocols [10], message-oriented middleware [11], and operating system schedulers [12]. The large number of performance-relevant parameters and their interdependencies complicate a manual model configuration. Therefore, automated support for design space exploration is desirable.

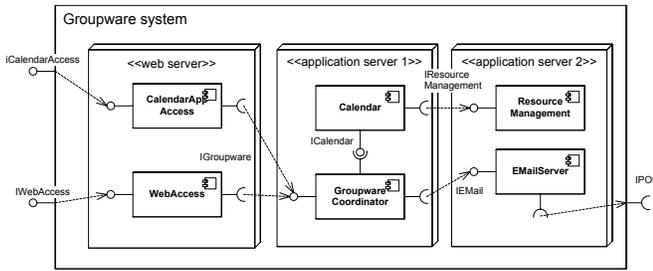


Fig. 2. Example system: Groupware

C. Terms

In the remainder of the paper, we will use several terms defined as follows:

- **Design Option:** An element in the architectural model, which could be changed to improve performance properties. This can include configuration options (e.g., thread-pool sizes), components selection, component allocation, or resource dimensioning.
- **Architecture Constraint:** A restriction defined on the architectural model, which specifies elements in the model that shall not be considered as design options. Such constraints can result from business requirements (e.g., the component of a partner company has to be used and cannot be substituted) or organisational constraints (e.g., only a certain type of operating system can be used due to available licences), or technical expertises (e.g., a certain design option is fixed, because the developers are familiar with it).
- **Architecture Candidate:** A variant of the architectural model, where a fixed set of configuration values has been chosen for each design option.

III. RUNNING EXAMPLE

We illustrate our design space exploration process using an example component-based architecture. Figure 2 shows the component-based groupware system architecture, and its allocation to servers. Users can plan meetings with multiple other participants and also schedule resources (rooms, projectors, etc). The systems checks dates for availability and suggests alternative dates if the original date is already blocked for one of the participants. Up to three alternative meeting dates including resource availability are checked. Upon appointing a meeting, e-mails are sent to the participants. Users can access the system using a web based access or the calender software on their local machine.

IV. PERFORMANCE-ORIENTED DESIGN SPACE EXPLORATION

The performance-oriented design space exploration fits in the development process model of component-based systems as introduced in section II-A. It extends the QoS analysis step, which was introduced in [5]. Within the QoS analysis step, the actual design space exploration takes place after having collected the required input data.

A. QoS Analysis Step

The extended *QoS-analysis* step, already sketched in section II-A, is shown in figure 3. The upper six workflows, that collect and integrate QoS information from various sources, are from [5]. The lower four workflows add specific information and execute the actual exploration of the design space.

QoS Information Collection and Integration (upper six workflows): The *domain expert* and the *system deployer* contribute their specific knowledge on system usage and system environment, respectively, to the architectural model. In our running example, the system deployer specifies the QoS characteristics such as CPU speed and hard disk drive speed of the three available servers for performance. Additionally, he specifies the deployment of the six components to the three servers as shown in figure 2. The domain expert derives the expected usage of the system from the use cases and creates refined usage models, which contain arrival rates of users as well as the requested services and the input parameters. In our example, he specifies that in average five users per minute will plan a meeting. The group size for a planned meeting is a performance relevant input parameter and the domain expert specifies a normal distribution between three and eleven participants.

The software architect formulates the QoS requirements of the system based on the business requirements. In our example, the planning of a meeting must not take longer than 30 seconds in 90% of all cases. Then, the software architect integrates the available information (namely annotated deployment diagram, annotated system architecture and refined usage model) to a fully QoS annotated architecture.

To enable performance-oriented design space exploration, we added the next four workflows to the QoS analysis step.

Architectural Constraint Specification: First, the software architect needs to specify constraints to mark what areas of the design space are not to be explored. The specific exclusion keeps the search within feasible regions. For example, the software architect can enforce the usage of certain already procured components, can restrict the number of available servers, or can mark certain design options as unchangeable since they are already decided for the project or the organisation. In our example, the software architect restricts the number of servers to the aforementioned three, because they have already been bought. Additionally, she specifies to use Remote Method Invocation (RMI) communication between the components, as this is the organisation-wide standard to use.

Design Option Specification: Next, the software architect can already provide additional architecture candidates. There are two reasons to manually specify candidates: (1) to include candidates into the analysis, that she knows to be an alternative, but that are too different to the original architecture (e.g. contain slight functional differences) to be found by the automated design space exploration, and (2) to provide more starting points for the exploration to speed up the analysis. For our example, we assume that the software architect does not provide any manual candidates and starts the design space

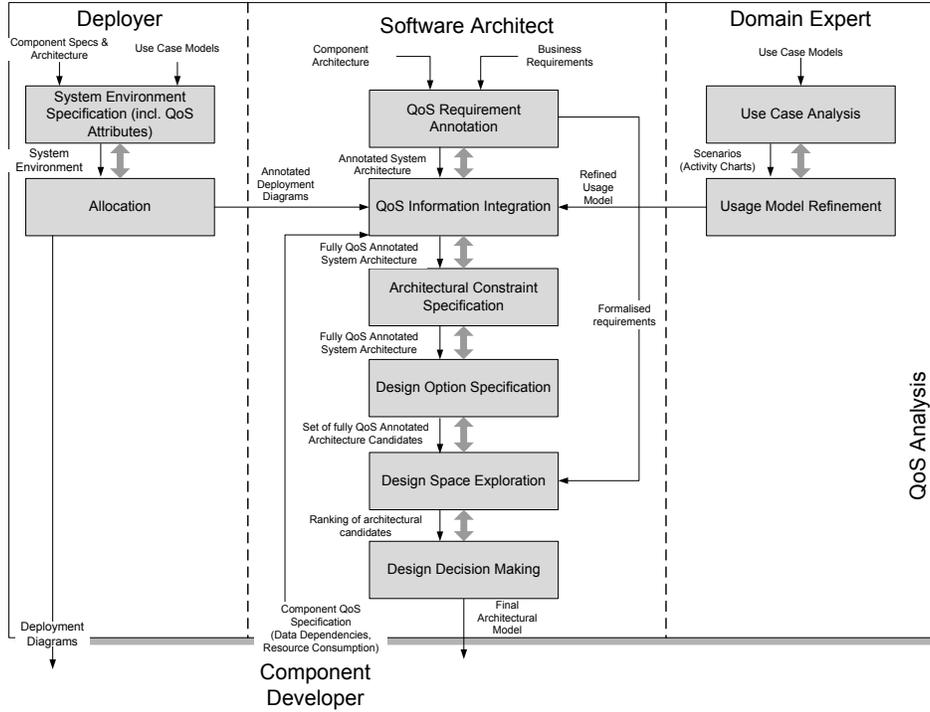


Fig. 3. QoS Analysis Step

exploration with just one candidate.

Design Space Exploration: After having specified a set of architecture candidates and the overall architecture constraints, the actual, semi-automated design space exploration can take place. Based on the set of initial architecture candidates, a tool generates further candidates and assesses them in respect to the requirements. The software architect might be involved at intermediate steps to decide whether generated candidates are feasible and to choose candidates for the final, detailed performance simulation. This step is explained in more detail in the next section IV-B. The output of this step is a (possibly empty) ranking of promising architectural candidates fulfilling the requirements.

Design Decision Making: Finally, the software architect decides for one of the candidates from the ranking. We make this step explicit and do not automatically use the candidate ranked best, because the software architect may have specific reasons not expressible in requirements for choosing a candidate that is inferior in the ranking.

B. Design Space Exploration Step

The goal of the design space exploration is to (semi-) automatically find a number of good architectural candidates satisfying the QoS requirements. Applying the design space exploration tool results in a ranking of architecture candidates based on their performance properties in the current project setting. Not just a single best solution is returned, so that the software architect can make the final decision which candidate is best.

Goal of the Search: Design space exploration is a search problem, which can only be solved heuristically for considerably large designs, as there are many architectural candidates. Two different options of the goal, i.e. two different stop criteria for the search in the design space exist:

- 1) the space is only searched until one (or a few) architecture candidates are found that satisfy the requirements (a constraint satisfaction problem),
- 2) the space is searched more thoroughly to find the architecture candidates with the best performance properties (an optimization problem)

The design space exploration process is identical for the two goal options, except that the stop criterion of the search is affected. Thus, from these two exploration goal options, the software architect can choose one that is important in her specific setting.

In our example, we let the tool search until we find one architecture candidate that satisfies the performance requirements.

Overview: The design space exploration workflow is depicted in figure 4. It starts with an initial set of architecture candidates fulfilling the functional requirements of the system. This initial set constitutes the initial population of the search. The search itself is an iterative process, in which in turn candidates are evaluated and new candidates are generated based on the evaluation, until the stop criteria are satisfied. A number of metaheuristic search techniques (cf. [13]) can be applied, such as genetic algorithms, simulated annealing, or random-restart hill climbing. As design space exploration is a search

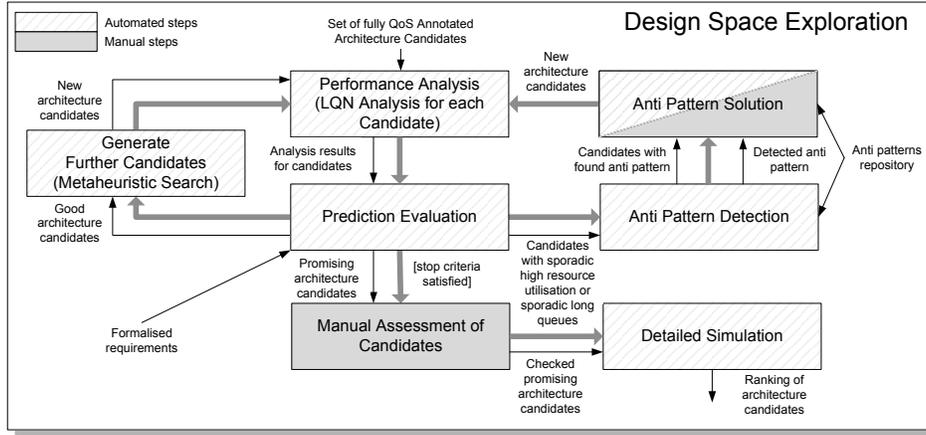


Fig. 4. Design Space Exploration Step

in a highly complex search space, the design space exploration workflow is computationally expensive [14]. However, speedy answers are not essential, because the approach is not applied to on-line, realtime software engineering problems (such as automated adaptation), but to software design.

Performance Analysis: In the first step, the performance of the candidates in the current population is analysed using a comparably fast performance analysis method (such as Layered Queuing Networks (LQNs) [15]) that results in mean values of performance metrics and additionally provides the standard deviation. Therefore, the QoS annotated system architecture model is automatically transformed in an analysis model. For our example, we transform the architecture model into an LQN (as demonstrated in [16] for the PCM). Then, the LQN can be solved using existing simulation tools.

Prediction Evaluation: Next, the tool compares the results of the analysis to the performance requirements of the system. Requirements concerning mean values of performance metrics can be directly compared. Requirements concerning percentiles (as in our example 90% of all cases, the planning of a meeting takes less than 30 seconds) can be roughly checked assuming a normal distributed response time and using the standard deviation of the results (e.g. from a simulation of LQNs). The assumption here is that the distributions are not too skewed. If a stop criterion is satisfied, the workflow advances to the manual assessment of candidates. As long as the stop criteria are not satisfied the search for better candidates continues.

Generate Candidates: To generate new candidates, we use design options specified in the performance meta-model (e.g. in the PCM [17]): Configuration options of components and middleware such as which communication protocol to use or the thread pool size of an application server, as well as allocation, resource dimensioning, and component selection. With all these different design options, a large design space is spanned with the different combinations, in which each design option is a dimension of change. There are two methods to generate new candidates based on the current population, which are

executed in parallel during the search, each contributing new candidates to the next iteration. The first method uses the classic metaheuristic search techniques to explore the design space (left in figure 4) and is the standard way to proceed. The second method uses the specific knowledge of the performance analysis to specifically solve performance problems in current candidates (e.g. the detection of a bottleneck resource), if applicable.

In the first method, the tool uses good candidates from the current population to generate new candidates. This depends on the search technique: Using genetic algorithms, the tool generates new candidates by combining candidates from the population, favourably those with good performance properties. Using hill climbing, the tool generates several new candidates by in each case changing a single design option of an already good candidate. Potentially, knowledge about the effects on performance of different design options can be integrated by adding a heuristic which design option to try first. Then, the tool evaluates newly generated candidates in the next iteration of the search.

In our example, hill climbing is used. As a first design option, a `ForkingEmailserver` component is available in the repository, which forks a separate thread for generating and sending e-mail via the POP interface. Thus, the control flow can immediately return to the `GroupwareCoordinator` component, reducing the waiting time for the user. However, more threads are required, so that this option is not always beneficial. The new candidate using the `ForkingEmailserver` component is depicted in figure 5 and referred to as candidate A in the following. Additionally, the tool generates a variety of further candidates B to E in the first iteration, each differing from the original candidate in one design option.

Anti-Pattern Detection and Solution In the second method, new candidates are generated by specifically solving performance problems detected in the current candidates. Performance metrics such as high resource utilisation and long queues are an initial indicator for the bottlenecks or perfor-

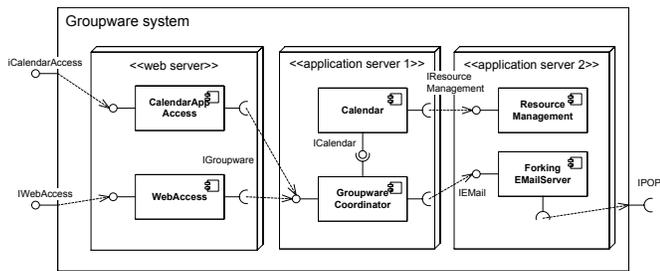


Fig. 5. Example: Architecture candidate A with an alternative Emailserver component

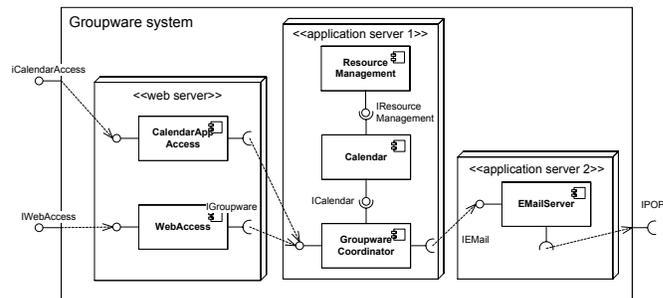


Fig. 6. Example: Architecture candidate F with changed allocation

mance anti-patterns in the design. For such candidates, the tool uses automated pattern detection to find the source of the problem, based on information stored in an anti-pattern repository. If a known anti-pattern or a single bottleneck resource is found, the corresponding solution can be applied. Many solutions, such as the replication of a resource in presence of a bottleneck, can be automated. However, not all solutions can be applied manually. For example, for a frequent communication between two remote component, a cache could be introduced to reduce the response time or the allocation could be changed. Because a performance model of the system usually does not contain enough information to decide whether a cache is applicable, the process requires manual interaction. Thus, the anti-pattern solution step is semi-automated.

In our example, the pattern detection finds excessive remote communication between the `Calendar` component and the `ResourceManagement` component, as the `Calendar` requests available resources from the `ResourceManagement` up to four times per planning request. First, the automated solution to change the allocation of the components is tried (depicted in figure 6 and later referred to as candidate F). Here, the `ResourceManagement` component is not coupled to any other component than the `Calendar`, thus, its allocation can be changed without creating new remote communication. However, now the application server 1 receives a higher workload. In future iterations of the process, we will see whether this actually improves the performance. To detect whether solutions have already been tried, the solution step keeps track of previously generated candidates. In our example, however, the application server 1 can handle the additional load, and the performance metrics improve due to less remote communication. If the problem persisted, the second semi-automated solution of introducing a cache would be suggested to the software architect.

Repetition: After having generated multiple new candidates using both methods in parallel, each new candidate of the new population is again analysed for performance and the results are again evaluated against the requirements and the stop criteria, forming the basis for generating new candidates.

In our example, a number of new candidates is created in the first iteration. Their performance is analysed in a repetition of the performance analysis step and compared to the requirements in the repetition of the prediction evaluation step.

In our case, candidates A and F now satisfy the performance requirements. As the goal in our example is to find a sufficient alternative, the search stops at this point, and the workflow proceeds to the manual assessment of candidates.

If the goal of the process was to find the optimal solution, the search would continue until no better candidates are found within a number of iterations. Depending on the search algorithm, new candidates would be generated based on already good ones. For the hill climbing search algorithm, the best of the current candidates would be used for further variation.

If no feasible solution is found after a certain amount of iterations, the tool also aborts the search. In that case, the software architect needs to manually improve the design, relax the constraints, or revisit the performance requirements.

Manual Assessment of Candidates: After the stop criteria are successfully satisfied, the tool presents the best architecture candidates to the software architect, who manually checks whether they fit all criteria. Possibly, the software architect detects that she missed to specify a beforehand implicit constraint. In this case, the software architect can revisit the constraint specification step (cf. section IV-A). She can also manually filter the best architecture candidates for this criterion and then proceed.

In our example, the software architect approves both candidates.

Detailed Simulation: As a last step, a detailed performance simulation (such as presented in [18]) is executed for the best architecture candidates. Here, the tool obtains not only mean values but distribution functions for the architecture candidates. As the simulation is comparably costly and takes several minutes for each architecture candidate, it cannot be integrated in the search iterations before. However, the resulting performance distributions allow to affirm more detailed performance requirements later on, that the tool has roughly checked before using mean response time and standard deviation.

In our example, the detailed simulation affirms that for both architectural candidates A and B, the planning of a meeting has a response time lower than 30 seconds in 90% of all cases. Thus, two architectural candidates leave the design exploration workflow. In the surrounding QoS analysis workflow, the software architect can now decide for one of the two options, or restart the exploration process with revised constraints or revised goals.

V. RELATED WORK

Classical performance analysis tools for queueing network or stochastic Petri-nets usually only produce performance metrics after analysing the model and do not provide feedback on how to improve the model. The SPE-ED tools by Smith et al. [1], which relies on queueing network analysis, features a visualisation of the performance metrics in the graphical representation of a performance model by colouring performance-critical steps. While this provides a starting point for improving the model, there are no concrete guidelines to make these steps less performance-critical.

Cortellessa et al. [19] propose an approach for automated feedback generation for software performance analysis, which aims at systematically evaluating performance prediction results using step-wise refinement. The approach relies on the (yet manual) detection of performance anti-patterns in the performance model. There is no support to automatically solve a detected anti-pattern, and there is no suggestion of new architecture candidates.

Bondarev et al. [20] introduce the DeepCompass framework for design space exploration of embedded systems. The framework relies on the ROBOCOP component model, and therefore supports CBSPE. It uses a Pareto analysis to resolve the conflicting goals of optimal performance and low costs for different architecture candidates. Therefore, performance metrics for each architecture candidate are plotted against the costs of each candidate. The approach requires a manual specification of all architecture candidates and provides no support for suggesting new candidates.

Parsons et al. [21] present a framework to detect performance anti-patterns in Java EE architectures. The method requires an implementation of a component-based system, which can be monitored for performance properties. It uses the monitoring data to construct a performance model of the system and then searches for EJB-specific performance anti-patterns in this model. This approach cannot be used for design space exploration in early development stages, but only to improve existing systems.

İpek et al. [22] describe an approach to automatically explore the design space of hardware architectures, such as multiprocessors or memory hierarchies. The authors automatically simulate multiple sample points in the design space and use the results to train a neural network. This network can be solved quickly for different architecture candidates and delivers accurate results with a prediction error of less than 5 percent. However, the approach does not reflect the specifics of software architectures and does not feature an architectural model.

Other than the former quantitative approaches, there are qualitative architectural evaluation methods, such as ATAM [23] or SAAM [24]. These approaches do not formally model software architectures, but instead rely on textual specification of usage scenarios. Therefore, these approaches only allow an informal discussion of performance properties and architecture candidates, but no automated support for

exploring the design space.

VI. OPEN ISSUES FOR DISCUSSION

There are numerous open issues, both for the realisation of the proposed approach and for future work.

- The **specification of requirements, architectural constraints and manual, initial design options** must be included in the architectural meta-model to allow the specification of these in the models and to consider them during the design space exploration. First, the performance requirements must be formalised so that the performance analysis results can be compared to them. Second, software architects must be able to specify architectural constraints. These can either be invariants (such as "component A must be used"), but also a definition of allowed parameter ranges or parameter limits (such as "the CPU speed must not exceed 2GHz"). Additionally, the software architect needs a way to specify design option in her initial architecture. Here, it is not desirable to copy the initial architecture and thus have two independent models, in one of which the design option is modelled. This way would lead to a hardly maintainable set of models. Thus, design options must be explicitly specified in the initial architecture, for example as a kind of architectural "diff".
- **Characteristics of the design space:** We need further insight into the characteristics of the design space of component-based software system's performance, i.e. explore the properties of search spaces. Search spaces only featuring a single (global and local) optimum are easier to handle than jagged search spaces with many local, but inferior optima. Only if we know the properties of the design space, we can choose an appropriate application of metaheuristic search techniques. Here, it is interesting to find out whether the performance-wise design space of component-based systems has typical characteristics over several systems under study, which would allow to choose well-fitting metaheuristic search techniques in our approach.
- **Multi-objective optimisation:** One important issues for future work is the handling of trade-off decisions. Usually, we do not want to optimise a system only for its performance metrics, as this would be far too expensive. At least two fitness functions of cost and performance are needed for useful exploration of the design space. Cost metrics can be easily integrated into the approach by assigning fix and variable cost to all elements of the models. Still, we are then faced with a multi-objective optimization problem: Candidates with good performance metrics and high costs cannot readily be compared to candidates with inferior performance metrics and lower costs. Here, we need to weight the different objectives or present the set of Pareto-optimal solutions (i.e. solutions that are superior to all others in at least one objective or equal in all objectives, cf. [3]) to the software architect for final decision. Next to cost, other quality properties

such as security, reliability and maintainability could also be integrated in the process, if evaluation methods are available.

- **Technology specific features:** Finally, it will be interesting to integrate technology specific features into the catalogue of available design options: For example, EJB application server configuration parameters could be integrated in the models to form explicit design options. If the impact of the parameters on performance is captured in middleware performance models (as planned for the Palladio approach, [17]), the resulting larger design space can be explored with the approach described in this paper. Additionally, technology-specific performance anti-pattern detection such as presented in [21] for EJB can be integrated in the approach.

VII. CONCLUSIONS

In this paper, we presented an approach for the (semi-) automatic, performance-oriented design space exploration of component-based software architectures. Our approach iteratively generates new architecture candidates based on an initial architecture and evaluates them for performance, using a mix of metaheuristic search techniques and anti-pattern detection. We illustrated the approach with a running example.

The first benefits of this approach is time-savings when the software architects encounters performance problems in a component-based software architectures and wants to explore the available design space. Second, the automated exploration can result in improved architectures, as many influence factors such as configuration parameters can be automatically taken into account, which the software architect might miss otherwise.

Further research includes the prototypical realisation of the approach and the empirical validation of the results as well as the benefits of using the approach. The choice of the right metaheuristic search technique (together with evaluation function, stop criteria and the choice of the next candidates) and the detection and solution of anti-patterns are critical points in this prototypical realisation.

REFERENCES

- [1] C. U. Smith and L. G. Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, 2002.
- [2] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, "Model-Based Performance Prediction in Software Development: A Survey," *IEEE Transactions on Software Engineering*, vol. 30, no. 5, pp. 295–310, May 2004.
- [3] M. Ehrgott, *Multicriteria Optimization*. Springer-Verlag, New York, USA, 2005.
- [4] J. Cheesman and J. Daniels, *UML Components: A Simple Process for Specifying Component-based Software*. Addison-Wesley, Reading, MA, USA, 2000.
- [5] H. Koziolok and J. Happe, "A Quality of Service Driven Development Process Model for Component-based Software Systems," in *Component-Based Software Engineering*, ser. Lecture Notes in Computer Science, I. Gorton, G. T. Heineman, I. Crnkovic, H. W. Schmidt, J. A. Stafford, C. A. Szyperski, and K. C. Wallnau, Eds., vol. 4063. Springer-Verlag, Berlin, Germany, July 2006, pp. 336–343. [Online]. Available: http://dx.doi.org/10.1007/11783565_25
- [6] S. Becker, H. Koziolok, and R. Reussner, "The Palladio Component Model for Model-Driven Performance Prediction," *Journal of Systems and Software*, 2008, accepted for publication, To Appear.
- [7] S. Balsamo, M. Marzolla, A. Di Marco, and P. Inverardi, "Experimenting different software architectures performance techniques: a case study," in *Proceedings of the fourth international workshop on Software and performance*. ACM Press, 2004, pp. 115–119.
- [8] H. Koziolok, S. Becker, and J. Happe, "Predicting the Performance of Component-based Software Architectures with different Usage Profiles," in *Proc. 3rd International Conference on the Quality of Software Architectures (QoSA'07)*, ser. LNCS, vol. 4880. Springer, July 2007, pp. 145–163.
- [9] H. Koziolok, S. Becker, J. Happe, and R. Reussner, *Model-Driven Software Development: Integrating Quality Assurance*. IDEA Group Inc., December 2008, ch. Evaluating Performance and Reliability of Software Architecture with the Palladio Component Model, p. To appear.
- [10] S. Becker, "Coupled Model Transformations," in *Proceedings of the 7th International Workshop on Software and Performance (WOSP2008)*. ACM Sigsoft, 2008, to Appear.
- [11] J. Happe, H. Friedrich, S. Becker, and R. H. Reussner, "A pattern-based performance completion for message-oriented middleware," in *WOSP '08: Proceedings of the 7th international Workshop on Software and Performance*, 2008, to appear.
- [12] J. Happe, "Concurrency Modelling for Performance and Reliability Prediction of Component-Based Software Architectures," Ph.D. dissertation, University of Oldenburg, 2008, to appear.
- [13] E. K. Burke and G. Kendall, Eds., *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Springer, 2005.
- [14] M. Harman, "The current state and future of search based software engineering," *Future of Software Engineering, 2007. FOSE '07*, pp. 342–357, May 23–25 2007.
- [15] G. Franks, "Performance analysis of distributed server systems," Ph.D. dissertation, Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada, December 1999.
- [16] H. Koziolok, "A Model Transformation from the Palladio Component Model to Layered Queueing Networks," in *Proceedings of the SPEC International Performance Evaluation Workshop 2008*, 2008, submitted.
- [17] S. Becker, "Coupled model transformations for qos enabled component-based software design," Ph.D. dissertation, University of Oldenburg, Germany, 2008, to Appear.
- [18] S. Becker, H. Koziolok, and R. Reussner, "Model-based Performance Prediction with the Palladio Component Model," in *Proceedings of the 6th International Workshop on Software and Performance (WOSP2007)*. ACM Sigsoft, February 5–8 2007.
- [19] V. Cortellessa and L. Frittella, "A framework for automated generation of architectural feedback from software performance analysis," in *EPEW*, ser. Lecture Notes in Computer Science, K. Wolter, Ed., vol. 4748. Springer, 2007, pp. 171–185. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-75211-0_13
- [20] E. Bondarev, M. R. V. Chaudron, and E. A. de Kock, "Exploring performance trade-offs of a JPEG decoder using the DeepCompass framework," in *WOSP '07: Proceedings of the 6th international workshop on Software and performance*. New York, NY, USA: ACM Press, 2007, pp. 153–163.
- [21] T. Parsons and J. Murphy, "Detecting performance antipatterns in component based enterprise systems," *Journal of Object Technology*, vol. 7, no. 3, pp. 55–90, Mar. 2008. [Online]. Available: http://jot.fm/issues/issue_2008_03/article01/
- [22] E. İpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz, "Efficiently exploring architectural design spaces via predictive modeling," in *ASPLOS*, J. P. Shen and M. Martonosi, Eds. ACM, 2006, pp. 195–206. [Online]. Available: <http://doi.acm.org/10.1145/1168857.1168882>
- [23] R. Kazman and L. Bass, "Categorizing business goals for software architectures," Carnegie Mellon University, Software Engineering Institute, CMU/SEI-2005-TR-021, Dec. 2005. [Online]. Available: <http://www.sei.cmu.edu/publications/documents/05.reports/05tr021.html><http://www.sei.cmu.edu/pub/documents/05.reports/pdf/05tr021.pdf>
- [24] R. Kazman, L. Bass, G. Abowd, and M. Webb, "SAAM: A method for analyzing the properties of software architectures," in *Proceedings of the 16th International Conference on Software Engineering*, B. Fadini, Ed. Sorrento, Italy: IEEE Computer Society Press, May 1994, pp. 81–90.