Automatically Improve Software Architecture Models for Performance, Reliability, and Costs Using Evolutionary Algorithms

Anne Martens*, Heiko Koziolek†, Steffen Becker‡, Ralf Reussner*‡
*Karlsruhe Institute of Technology, Karlsruhe, Germany
Email: {martens, reussner}@ipd.uka.de
†ABB Corporate Research, Ladenburg, Germany
Email: heiko.koziolek@de.abb.com
‡FZI Karlsruhe, Karlsruhe, Germany
Email: {becker, reussner}@fzi.de

ABSTRACT
Quantitative prediction of quality properties (i.e., extra-functional properties such as performance, reliability, and costs) of software architectures during design supports a systematic software engineering approach. Designing architectures that exhibit a good trade-off between multiple quality criteria is hard, because even after a functional design has been created, many remaining degrees of freedom in the software architecture span a large, discontinuous design space. In current practice, software architects try to find solutions manually, which is time-consuming, can be error-prone and can lead to suboptimal designs. We propose an automated approach to search the design space for good solutions. Starting with a given initial architectural model, the approach iteratively modifies and evaluates architectural models. Our approach applies a multi-criteria genetic algorithm to software architectures modelled with the Palladio Component Model. It supports quantitative performance, reliability, and costs prediction and can be extended to other quantitative quality criteria of software architectures. We validate the applicability of our approach by applying it to an architecture model of a component-based business information system and analyse its quality criteria trade-offs by automatically investigating more than 1200 alternative designs candidates.

Categories and Subject Descriptors
C.4 [Computer Systems Organization]: Performance of Systems—modelling techniques; D.2.8 [Software Engineering]: Metrics—performance measures; D.2.11 [Software Engineering]: Software Architecture

1. INTRODUCTION

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WOSP/SIPEW 2010, San Francisco Bay Area, USA
Copyright 200X ACM X-XXXX-XX-X/XXX ...$10.00.

One benefit of modelling software architectures is the ability to quantitatively analyse extra-functional properties, such as performance or reliability, based on the software architecture model during early design stages. This approach avoids costs for late-life cycle performance/reliability fixes and architectural redesigns. Several methods (e.g., UML [31], SySML [32], LQN [16], Palladio [2]) allow to model a software architecture including static structure, dynamic behaviour and allocation to resources. Such models can be annotated with estimated or measured quality annotations (e.g., UML MARTE [30] for performance) and analysed for the different quality properties. For example, performance can be analysed using numerical techniques from queueing theory or simulation.

However, while many approaches allow to predict single quality properties of a given design, they do not support the software architect if the predictions indicate a violation of the requirements. For performance, the software architect needs to understand the performance prediction results, manually identify the root cause for the insufficient performance and manually determine an improved architecture model. An improved architecture model can for example contain faster / more reliable software components, a more powerful hardware environment, or a different allocation of components to hardware nodes. Even if requirements are fulfilled, there might be potential for improvement in the software architecture to save costs by reducing over-provisioned resources. In general, improving one quality property can deteriorate another, thus, quality properties cannot be improved in isolation.

Interpretation of prediction results, problem identification, and improvement of the software architecture are manual tasks in current practice [35]. Automation is desirable, because the manual tasks (i) require vast architectural experience, (ii) are laborious and therefore cost-intensive, and (iii) are error-prone due to the overwhelmingly complex design space for human beings.

Researchers have proposed rule-based and metaheuristic-based solution approaches for the problem of automatically finding architectural designs with improved performance or reliability properties. Rule-based approaches (e.g., [38, 11, 33, 28] try to identify problems in the model (e.g., bottle-
necks) based on predefined rules and apply predefined solutions for these problems. Existing rule-based approaches only focus on performance analysis without considering other quality criteria. They operate on the performance model instead of the architectural model and are therefore difficult to use for regular architects not familiar with performance formalisms. These approaches cannot find solutions for which no rule exists, thus, they cannot cover all possible solutions and might result in locally optimal solutions. Metaheuristic-based approaches (e.g., [19, 18, 7]) encode the challenge of improving architectures as an optimization problem and use metaheuristic search techniques [3, 22] (e.g., genetic algorithms, simulated annealing, etc.) to find better design models. Existing metaheuristic approaches do not support performance prediction and only study the effect on reliability and costs of replicating components.

In this work, we present an approach which is capable to automatically improve a given architecture model with respect to performance, reliability, and cost. The approach is best suited for component-based software architectures. Components encapsulate functionality that can be independently reused, and thus component-based software architectures provide degrees of freedom to be exploited by our approach. In particular, we model architectures with the Palladio Component Model (PCM) [2] in this work. Quality property prediction is done using Layered Queueing Networks (LQN) [16] (or SimuCom EQNs [2]) for performance metrics; Markov models for reliability metrics [25]; and a newly introduced PCM costs extension for costs. The approach automatically searches the design space starting from a given initial architectural model. Using existing multi-criteria evolutionary algorithms, it iteratively modifies and evaluates architecture models. It searches for Pareto-optimal [14] architecture models with respect to performance, reliability, and costs. As metaheuristics are used, it is a best-effort approach and cannot guarantee global optimality of the results.

With our approach, software architects do not have to search for alternative solutions manually. Instead, they can focus on good solutions automatically determined by our approach for trade-off decisions between multiple quality criteria. As the approach works on the architectural model level (as opposed to the performance model), architects can directly understand and use the automatically found solutions.

The contribution of this paper is a new approach to find architectural design models with optimal performance, reliability, and costs properties given an initial architectural model. Other than former approaches, our approach works on the architectural level and does not require knowledge on quality criteria analysis models and methods (e.g., queueing theory knowledge for performance). We have implemented our approach in the PerOpteryx tool\(^1\). We present a case study for a business reporting system (12 components, 40 tasks in LQN).

This paper is organized as follows. Section 2 compares related approaches to our approach. Section 3 introduces a running example to explain the problem in detail as well as the used architecture model and quality prediction techniques. Section 4 describes the optimisation process. The case study is presented in Section 5. Finally, Section 6 discusses assumptions and limitations before Section 7 concludes.

2. RELATED WORK

Our approach is based on software performance prediction [34, 1], architecture-based software reliability analysis [17], and search-based software engineering [22]. We categorize closely related approaches into (i) rule-based approaches and (ii) metaheuristic-based approaches.

Rule-based Approaches: Xu et al. [38] present a semi-automated approach to find configuration and design improvement on the model level. Based on a LQN model, performance problems (e.g., bottlenecks, long paths) are identified in a first step. Then, rules containing performance knowledge are applied to the detected problems. The approach cannot detect improvement for which no rules have been defined, and some of the suggested improvements require changing the implementation of components, which is not desired when dealing with black box components. As the approach suggests changes on the level of LQNs, it might not be feasible to map suggested solutions back to the design. For example, it might be impossible to speed-up a certain component implementation to reach a certain service time because of inherent algorithmic complexity.

McGregor et al. [28] have developed the ArchE framework. ArchE assists the software architect during the design to create architectures that meet quality requirements. It helps to create architectural models, collects requirements (in form of scenarios), collects the information needed to analyse the quality criteria for the requirements, provides the evaluation tools for modifiability or performance analysis, and suggests improvements. Compared to our work, ArchE does not search the whole design space, but advances step-wise based on rules. It does not support reliability and only features a simple performance model. The architecture model is not component-based, consequently, degrees of freedom as presented later in this paper cannot be readily identified.

Cortellessa et al. [11] 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 common performance problem patterns (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. [5] introduce the DeepCompass framework for design space exploration of embedded systems. The framework relies on the ROBOCOP component model. 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. [33] present a framework for detecting 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 per-
formance antipatterns in this model. This approach cannot be used for design space exploration in early development stages, but only to improve existing systems.

**Metaheuristic-based Approaches:** Grünske [18] studies the improvement of two quality criteria, namely reliability and costs, to allow well-informed trade-off decisions. Evolutionary computing is applied to search for good design solutions. However, only redundancy of components is studied as a degree of freedom to improve reliability. Our approach envisions many degrees of freedom (three of which we present in this work). In [19], the same author presents a framework to predict values for multiple arbitrary quality criteria. However, this work does not consider optimisation or improvement yet.

Canfora et al. [7] optimise service composition costs using genetic algorithms while satisfying SLA constraints. Services are assumed to have fixed performance metrics that do not change for changing composition. Only service selection is considered as a degree of freedom, and trade-offs with other quality criteria are not considered.

Kavimandan et al. [24] presents an approach to optimise component allocation in the context of distributed real-time embedded component-based systems. They enhance pre-existing bin packing algorithms with the consideration of component consideration and deploy components together that have a compatible configuration. In total, only allocation is considered as a degree of freedom, but the authors also mention that their approach could be combined with other approaches.

3. BACKGROUND AND RUNNING EXAMPLE

To quickly convey our contributed concepts to the reader, we provide a running example in the following and introduce the existing quality analyses methods our approach is based on.

Our approach requires an (preferably component-based) architecture model with performance, reliability, and costs annotations as input. Balsamo et al. [1] have surveyed many different methods for specifying performance models and Smith and Williams [34] have provided a set of guidelines on how to obtain performance models during early development stages and on how to refine such models as the implementation progresses. UML2 extended with the MARTE profile is a popular example for a performance modelling notation. A survey on reliability prediction approaches is given in [17].

We have implemented our approach based on the Palladio Component Model (PCM). In principle, our approach could be applied on most known architectural performance and reliability models and analysis methods. The PCM is beneficial for our purposes as it is specifically designed for component-based systems. It strictly separates parametrised component performance models from the composition models and resource models, and it provides configuration options of the models. Thus, the PCM naturally supports many architectural degrees of freedom (e.g., substituting components, changing component allocation, etc.) to be exploited by our approach. Additionally, the model-driven capabilities of the PCM allow an easy generation of alternative architecture candidates.

Consider the minimal PCM model example in Fig. 1, which is realised using the Ecore-based PCM metamodel and visualized here in UML-like diagrams for quick comprehension. The system model specified by the software architecture consists of three connected software components C1 - C3 deployed on three different hardware nodes. The software components contain costs annotations, while the hardware nodes contain annotations for performance (processing rates), reliability (mean time to failure (MTTF), mean time to repair (MTTR)), and costs (fixed and variable costs in an abstract costs unit).

![Figure 1: Simple Example PCM Model](image)

For each software component service, the component developers provide an abstract behavioural description called service effect specification (SEFF). SEFFs model the abstract control flow through a component service in terms of internal actions (i.e., resource demands accessing the underlying hardware) and external calls (i.e., accessing connected components). Modelling each component behaviour with separate SEFFs enables us to quickly exchange component specifications without the need to manually change system-wide behaviour specifications (as required in e.g., UML sequence diagrams).

Component developers can specify resource demands for their components (e.g., in terms of CPU instructions to be executed), which can then be divided by the processing rate of the modelled resource environment to determine the actual execution time demanded from the processors. They can also specify failure probabilities for component internal actions, which can be determined for example using software reliability growth models [29], or code coverage metrics during testing. The PCM also supports hard disk drive rates and software resources such as thread pools.
A software architect composes component specifications by various component developers into an application model. With an additional usage model describing user arrival rates (open workload) or user population and think time (closed workload) of the system, and an additional model of the resource environment and the allocation of components to resources, the model is complete and can be transformed into analytical or simulation-based models for quality analyses. For the PCM, we briefly explain the analysis methods for the three considered quality criteria performance, reliability, and costs. For each architectural candidate, we evaluate the quality property (e.g., “response time 5 sec”) for each quality criterion (e.g., criterion “performance” with the metric “response time of a service”).

- **Performance**: For performance analysis, the PCM supports a transformation into a discrete-event simulation (SimuCom [2]) or LQNs to derive response times, throughputs, and resource utilizations. SimuCom is based on model-to-code transformations and the SSJ simulation framework. It is in the class of extended queueing networks and allows analysing models containing resource demands specified as arbitrary distribution functions, but can be time-consuming to derive stable results. The LQN transformation PCM2LQN [26] generates an LQN instance from a PCM instance. The LQN solver [16] provides a heuristic performance analysis and can often quickly produce results. However, it does only support means values as prediction results and does not support arbitrary distribution functions. In the remainder of this paper, we report results obtained using the LQN solver.

- **Reliability**: For reliability analysis, the PCM supports a transformation into absorbing discrete time Markov chains (DTMC) [25] or a reliability simulation to derive the probability of failure on demand (POFOD) for a usage scenario. The Markov chain generator combines all behavioural models and creates failure states for hardware resources based on the MTTF / MTTR values in a PCM instance. The reliability simulator is based on SimuCom and emulates the execution of the system. It generates exceptions based on the failure probabilities specified in the PCM and simulates hardware crashes based on the MTTF / MTTR values of modelled hardware resources. In the remainder of this paper, we report results obtained using the analytic DTMC solver.

- **Costs**: For costs analysis, we have developed a PCM costs solver for this paper. It relies on a static analysis of a PCM model instance. It calculates the costs for each component and resource based on the annotations specified in the PCM instance and then adds these costs to derive the overall expected costs for the architecture. Costs can include both initial and operational costs. If a server specified in the model is not used, i.e. no components are allocated to it, its costs do not add to the overall costs. The goal of this simplistic model is to allow costs to be considered, not to provide a sophisticated costs estimation technique. For the latter, existing costs estimation techniques such as COCOMO II [4] could be integrated here to obtain more accurate values.

The predicted quality properties for the example shown in Fig. 1 are depicted in Tab. 1. Although the example in Fig. 1 is simple, it is not obvious how to change the architectural model efficiently to improve the quality properties. For example, the software architect could increase the processing rate of server S1, which would result in better performance but higher costs. The software architect could also change the component allocation (3! = 27 possibilities) or incorporate other component specifications with different QoS attributes.

<table>
<thead>
<tr>
<th>Quality Criterion</th>
<th>Metric</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>Avg. Resp. Time</td>
<td>4.6</td>
<td>sec</td>
</tr>
<tr>
<td></td>
<td>Utilization S1</td>
<td>42</td>
<td>%</td>
</tr>
<tr>
<td></td>
<td>Utilization S2</td>
<td>37</td>
<td>%</td>
</tr>
<tr>
<td>Utilization S3</td>
<td></td>
<td>10</td>
<td>%</td>
</tr>
<tr>
<td>Reliability</td>
<td>POFOD</td>
<td>7.36E-4</td>
<td>%</td>
</tr>
<tr>
<td>Costs</td>
<td>Overall Costs</td>
<td>54</td>
<td>units</td>
</tr>
</tbody>
</table>

Table 1: Quality Property Prediction Results

The design space even for such a simple example is huge. Manually checking the possible design alternatives in a trial-and-error approach is laborious and error-prone. The software architect cannot easily create design alternatives that are even locally optimal for all quality criteria, and finding global optima is practically impossible because it requires modelling each alternative. In practice, this situation is often mitigated by overprovisioning (i.e., incorporating fast and expensive hardware resources), which can lead to unnecessary high costs.

4. **OPTIMISATION PROCESS**

To automatically improve software architectural models for performance, reliability and costs, we propose an automated optimisation process that takes an initial architectural model as input and searches for Pareto-optimal candidate solutions. With the results of our process, software architects can focus on good solutions for their trade-off decisions between multiple quality criteria.

To present our optimisation process, we first give an overview on the process in Section 4.1. Then, we describe in detail the considered degrees of freedom (Section 4.2), the resulting search problem formulation (Section 4.3) and the evolutionary optimisation (Section 4.4).

4.1 **Overview and Example**

Figure 2 shows an overview of our optimisation process. Input is an initial architectural model of the system, named initial candidate. In our case, this is a complete PCM model instance as shown in Figure 1. In addition, generic degrees of freedom to be considered by the approach must be defined in advance. An example generic degree of freedom is components reallocation to other servers. The optimisation process starts with the search problem formulation. The initial candidate is analysed for occurrences of the generic degrees of freedom, resulting in system-specific degrees of freedom. An example system-specific degree of freedom is allocating component C1 to server S1, S2 or S3.

In the second step, evolutionary algorithms iteratively search for better solutions. Applying the main steps of (a) reproduction, (b) evaluation, and (c) selection to a population of candidate solutions in each iteration.
4.2 Generic Degrees of Freedom

The system cannot be changed arbitrarily in an automated approach. A human designer cannot be replaced. But to our advantage, software architectural models contain a number of explicit degrees of freedom for variation that affect quality properties, but do not affect the functionality of the system. In particular, we only consider degrees of freedom that do not modify the interfaces used in the architecture. We further assume that components providing the same interface provide the same functionality. From this it follows that the variation along the considered degrees of freedom retains functionality.

In the following, we present the generic degrees of freedom that we have identified so far in software architectural models and which can be exploited by our approach. We start our list with four generic degrees of freedom that are already exploited by our current optimisation process implementation. We continue with three degrees of freedom currently available in the PCM and similar models but not yet exploited in our implementation. Finally, we present further three envisioned degrees of freedoms.

Allocation of components to servers can be changed, this is an integral part of most performance-prediction models and has large effects on the performance of a system.

Processing rates of the available hardware resources (CPU, HDD, …) can be changed in a certain range. Here, a discrete set of CPU types all having a defined processing rate and costs could be modelled. Currently, we model processing rates as a continuous range, assuming that a CPU model for virtually any processing rate within a range is available (e.g. by considering energy saving states (P-state) with resulting lower costs).

Number of servers: Servers can be added to provide more processing capacity or can be removed to save costs.

Selection of components and services: If functionally-equivalent components with different non-functional properties are available, they can be exchanged. Currently, we deem that a component B can replace a component A if B provides (i.e. implements) all interfaces provided by A and if B requires at most the interfaces required by A. Similarly, external services such as web services in a SOA system can be exchanged.

Component replication: In addition to servers, software components can be replicated and distributed to several servers. For example, large scale web applications will have several web servers and application servers. Here, constraints can be formulated for software components that are not ready for replication, for example because of component state.

Component configuration parameters: If components and their performance models provide configuration parameters, their values can be varied during the search. For example, a component can have a parameter to choose from several available compression algorithms.

Passive resources multiplicity, such as thread pool size or database connection pool size, can be varied to find a good balance for the utilisation of underlying resources. Of course, multiplicity of locks for mutual exclusion regions (which can also be modelled with passive resources with capacity of 1) must not be varied by our approach.

Further configuration of the software stack could be available in the architectural model. Models have been proposed for operating system (OS) scheduler configuration [20], message-oriented-middleware configuration [21]; and can be envisioned for other configurable OS properties or JVM configuration of for example garbage collection. In addition, selection of software stack elements can be explored by the optimisation if models are available: JVM selection (Sun’s JVM, Oracle’s JRockit JVM, …), OS selection (Windows, Linux, …), possibly also the choice of hypervisors in virtualised environments. These options can either be explicitly provided in software architectural model as suggested by [23], or their effects can be added to the performance model as completions, as suggested by [36] and realised in e.g. [21].

Priorities: For several usage scenarios with different quality requirements, the system could prioritise requests with tight response time requirements. For example, business-relevant transactions can be assigned a higher priority than maintenance functions. Priority optimisation as presented by [15] could be included here.

Custom degrees of freedom could be specified by the software architect by annotating any model element with a range of possible values and possibly the related performance, reliability and costs effects. For example, if a component-internal algorithm is designed, software architects could identify possibilities for tuning the algorithm, resulting in lower resource demand of a component’s internal action but higher costs of the component. A language for specifying such custom degrees of freedom would be required.

In any case, to exploit a degree of freedom, it has to be available in the software architectural model and meaningful transformations to performance models, reliability models and for costs calculation have to be available. For our running example, we consider three generic degrees of freedom in the following: 1. “Change CPU speed of a server”, 2. “Component allocation” and 3. “Component selection”.

4.3 Search Problem Formulation

As a first step in the optimisation process for a specific system, we analyse which system-specific degrees of freedom are available in the given software model based on the given general degrees of freedom. For the three generic degrees
Table 2: Degrees of freedom in the example

<table>
<thead>
<tr>
<th>Component selection</th>
<th>Alternatives for C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>(C2, C2a)</td>
</tr>
</tbody>
</table>

Table 2: Degrees of freedom in the example

of freedom used in our motivating example, seven system-specific degrees of freedom can be identified as shown in Table 2. We identify three system-specific degrees of freedom to change a processing speed of a CPU, one for each concrete CPU in the model, as well as three allocation degrees. For the third generic degree of freedom, assume a fourth available component C2a, that is faster, more reliable but also more expensive than C2, exists as shown in Figure 3. Then, for the component selection, one system specific degree of freedom exists for C2, whereas C1 and C3 have no alternatives available.

Figure 3: Alternative Implementation of C2

For each system-specific degree of freedom, we can determine a range of options. For example, the system-specific degree of freedom of where to allocate C1 has the three options {S1, S2, S3}. CPU speed of available CPU models might range from 25E+9 to 50E+9 instructions / sec.

In general, for each system-specific degree of freedom \(D_i\) (with an index set \(I\) and \(i \in I\)), we can derive the set of possible choices: the design option set \(O_i\), as exemplary shown in the last column of Table 2. Constraints on the system-specific degrees of freedom can be formulated at this point by reducing the design option set of a system-specific degree of freedom. For example, it could be excluded that C1 can be allocated to S3 in our example, resulting in the remaining options \{S1, S2\}.

The design space \(DS\) in which the search problem is the Cartesian product of the design option sets of all system-specific degrees of freedom: \(DS = \Pi_{i \in I} O_i\). The design space \(DS\) is at the same time the set of all possible candidate solutions. Each candidate \(c\) can be expressed as a vector of chosen design options: \(c \in DS\) with \(c = [o_1 \in O_1, ..., o_n \in O_n], n = |I|\). In our example, the initial candidate \(c_0\) can be expressed as \([35, 40, 30, S1, S2, S3, C2]\) with the ordering of degrees of freedom as given in Table 2 and omitting the \(E\) notation part.

With these definitions, the problem is suited for evolutionary algorithms. A candidate is represented by a vector of chosen design options, which is named the candidate’s genotype. The genotype is varied in the reproduction phase, where it is ensured that each gene has a value within the design option set. Based on a candidate \(c\)'s genotype, a software architecture model for \(c\) can be derived by inserting the chosen design options in the initial candidate model. The resulting model is named the phenotype of \(c\). For this phenotype of \(c\), quality properties can be predicted in the evaluation phase, in our case with the PCM.

With this problem formulation, every newly generated phenotype is valid, as long as there are no further constraints on combinations of degrees of freedom. An example for a constraint on combinations is that C1 and C2 must not be deployed on the same server because of e.g. conflicting system library version requirements. This does not limit the design options for each degree of freedom, but does constrain the set of all candidates \(DS\). Checks must be made in the selection phase of the metaheuristic, or the reproduction phase has to take constraints into account. Note that such a constraint is only formulated if a candidate is (1) functionally infeasible or (2) infeasible for quality criteria that cannot be automatically quantitatively evaluated, such as maintainability.

4.4 Evolutionary Optimisation

Candidate solutions can be evaluated for optimal trade-offs, i.e. for Pareto-optimality. A candidate architecture is Pareto-optimal, if it is superior to any candidates evaluated so far in at least one quality criterion. More formally: Let \(a\) be a candidate solution, let \(DS\) be the set of all possible candidates, let \(C \subseteq DS\) be a set of candidate solutions evaluated so far, and let \(q\) be a quality criterion with a domain \(D_q\), an evaluation function \(f_q : C \rightarrow D_q\) so that \(f_q(c)\) denotes the quality property of a \(c \in C\) for the quality criterion \(q\), and an order \(\leq_q\) on \(D_q\) so that \(c_1 \leq_q c_2\) means that \(c_1\) is better than or equal to \(c_2\) with respect to quality criterion \(q\). Then, candidate solution \(a\) is Pareto-optimal with respect to a set of evaluated candidate solutions \(C\), if

\[
\forall b \in C \exists q : f_q(a) \leq_q f_q(b)
\]

If a candidate solution is not Pareto-optimal, then it is Pareto-dominated by at least one other candidate solution in \(C\) that is better or equal in all quality criteria. Analogously, a candidate is globally Pareto-optimal, if it is Pareto-optimal with respect to the set of all possible candidates \(DS\).

The optimisation problem can be formulated as follows for a set of quality criteria \(Q = \{q_1, ..., q_n\}\):

\[
\min_{c \in DS} [f_{q_1}(c), ..., f_{q_n}(c)]
\]

Constraints on combinations can be added to the optimisation problem at this point.

For the optimisation, we face a non-linear combinatorial optimisation problem [3, Def. 1.1]. We cannot apply classic techniques such as Branch-And-Bound [12]. Metaheuristics have been successfully applied to similar problems in software engineering [22]. In this work, we use evolutionary algorithms, as for example described in [3, p. 284], as they are found useful for multi-objective problems [9]. Other metaheuristics could be used as well.

We chose not to use a rule-based approach (which is a local search technique) as other work do [35, 11, 33, 28], as they are restricted to limited degrees of freedom each. No a-priori knowledge about the effects of many of the degrees of
freedom is available. For example, rules for the exchange of components would require a numerical solution for optimal component composition, which is not possible in general because of the parametrisation of the component SEFFs. For other degrees of freedom, such as allocation, rules can give guidance, but cannot foresee the complexity of performance metrics introduced by software resources and contention effects. For example, passive resources such as thread pools are involved, allocation of components to servers cannot be solved by a bin-packing algorithm based on the resource demand of components only. Metaheuristic can search regions of the search space for which no prior knowledge about the relation between choices and resulting quality properties exists. Only a quantitative evaluation function for each quality criterion based on an architectural model is required.

Figure 4 shows the process model of our method. The method is described here exemplary for our current realisation with the PCM and the NSGA-II evolutionary algorithm [13] as implemented in the Opt4J framework [27], but can as well be used for other software architecture modelling languages and other population-based metaheuristic search techniques. The process starts with an initial model of a component-based software architecture (initial candidate) and modifies it along the system-specific degrees of freedom. As the software model contains all required annotations, all steps of the search can be completely automated. Figure 5 shows the first two iterations of an exemplary run, starting with an initial given candidate \( c_0 \) (we left the not-notation out and rounded to E+9 for brevity).

Reproduction: Based on the currently available candidates in the population, new candidate solutions are derived by “mutation” or “cross-over” or they are randomly created. With mutation, one or several design options are varied. In our exemplary run, based on the initial candidate \( c_0 \), a new candidate \( c_2 \) with changed allocation is derived in the first iteration that allocates C3 on S2. Candidate \( c_7 \) derives from \( c_3 \) in the second iteration by reallocating C2a to S1. With cross-over, the genotypes of two good candidate solutions are merged into one, by taking some of each candidates design option values for the cross-over. For example, candidate \( c_2 \) and candidate \( c_3 \) are combined by cross-over in the second iteration to produce \( c_6 \). In addition, candidates can be randomly generated based on the available design options. For example, candidate \( c_1 \) is randomly created here in the first iteration.

Performance domain specific heuristics could be integrated here to guide the search. For example, a heuristic mutation could move a component from an over-utilised server to a lightly utilised server.

Evaluation: In the second step, each newly derived candidate is evaluated for each quality criterion of interest. In our case, performance, reliability and costs metrics are predicted as described in sections 3. As a result, each candidate is annotated with the determined QoS properties. In our example, candidates \( c_1 \) to \( c_4 \) are evaluated in the first iteration, and candidates \( c_5 \) to \( c_8 \) are evaluated in the second iteration. Candidate \( c_9 \) has been evaluated before the actual optimisation starts. The results for each candidate are depicted in Figure 7 in the next section.

Selection: After the reproduction phase, the population has grown. In the selection phase, the population is again reduced by removing less promising candidates. Here, we first remove candidates that violate quality requirements. Second, we filter Pareto-dominated candidates and only keep Pareto-optimal ones. In our example, candidates \( c_0 \), \( c_1 \) and \( c_4 \) are removed in the selection phase of iteration 1, and candidates \( c_5 \) and \( c_7 \) are removed in the selection phase of iteration 2. Possible future selection strategies could also keep a number of dominated or violating candidates to keep a diverse “gene pool” and avoid over-specialisation in local optima.

Over several iterations, the combination of reproduction and selection lets the population converge towards the front of globally Pareto-optimal solutions. If the search also keeps a good diversity of candidates, we can find solutions near
Section 5.1 Business Reporting System

The system under study is the so-called business reporting system (BRS), which lets users retrieve reports and statistical data about running business processes from a data base. It is loosely based on a real system [37]. Fig. 8 shows some parts of the PCM instance of the BRS visualised using annotated UML diagrams. It is a 4-tier system consisting of several software components. The WebServer component handles user requests for generating reports or viewing the plain data logged by the system. It delegates the requests to a Dispatcher component, which in turn distributes the requests to four replicated ReportingServers. The replication helps balancing the load in the system, because the processing load for generating reports from the database contents is considered significant. Each ReportingServer component is a composite component consisting of an inner ReportingEngine and a Cache component. The ReportingServers access two replicated Databases for the business data.

Besides the static view of the system, Fig. 8 also contains a behavioural view in form of service effect specifications in the upper half of the figure. The SEFFs contain the resource demands, failure probabilities, and call propagations later predictions will be based on. The components are allocated on eight different servers connected by different network devices. Our case study analyses a usage scenario, where 50 users access the system concurrently. Each user requests a report or view from the system and then looks at the results for 10 seconds before issuing the next request (closed workload).

For the performance prediction, we transform the model into a LQN (Fig. 9) using PCM2LQN [26]. The LQN model contains multiple dummy processors and tasks, which were added to make the transformation from PCM more straightforward. The average response time for this model as predicted by the LQNS tool is 2.2 seconds.

For the reliability prediction, we use the PCM Markov translator [6], which predicts a probability of failure on demand for the system of 0.0605 percent. This means that each user request will be successful with a probability of 99.9395 percent. The costs for the system calculated by the PCM costs solver are 98 units.

5.2 Search Problem Formulation

To formulate the search problem for the business reporting system, first the system specific degrees of freedom have to be determined. For our case study, we consider the following degrees of freedom: component selection, server processing rates, and component allocation.

Component selection is possible in this system as it con-
contains several replaceable standard components. The Web Server as well as the Database can be realised using third party components. The software architect can choose among multiple functional equivalent components with different non-functional properties and costs. For the BRS, we have modelled two additional web servers and one additional database which have different performance and reliability properties, but also higher or lower costs than the components in the initial system.

Server processing rates can be adjusted at multiple locations in the model as it contains 8 servers. It is expected that the overall performance of the system increases most significantly when using faster processing rates for highly utilised components. We assume here that the bounds for the processing rate are 1/2 of the initial rate (lower bound) and 2 times the initial rate (upper bound). Currently, the processing rate is modelled as a continuous variable.

Component allocation can be crucial for the non-functional properties and costs of the system. It could be possible to allocate multiple components on the same server without affecting the performance or reliability significantly. This could allow to remove some servers to save costs.

The genome of the initial candidate determined by PerOpteryx looks as follows: [1, 1, 1, 1, 1, 1, 1, 1, 1, DB, WebServer, DB, server1, server2, server4a, server4b, server3a, server3b, server3c, server3d]. It reflects the processing rates, the alternatively used components as well as the component allocation to different servers (e.g., WebServer is deployed on server 1 and Dispatcher is deployed on server 2).
5.3 Evolutionary Optimisation

For the evolutionary optimisation of the model, our prototype PerOpteryx tool follows the process described in Section 4.4. We configured Opt4J to run for 40 iterations and to produce 60 candidates per iteration. The LQN solver was configured with a convergence value of 0.001 and an iteration limit of 200 (see [26] for details). The PCM Markov model solver was used in a standard configuration.

The automatic improvement process took 8 hours, produced more than 1235 valid architectural candidates and made performance, reliability, and costs predictions for them. The average creation, transformation, and prediction time per candidate was 24 seconds. 58 of the candidates were deemed Pareto-optimal by the evolutionary algorithm.

The evolution process run time could be shortened significantly by executing the candidate analyses per candidate concurrently (e.g., on multicore processors or in a distributed environment). We consider this enhancement to our tool as future work.

5.4 Present Results

The results of the evolutionary optimisation run are depicted in Figures 10 to 12.

Figure 10 visualises the three dimensional Pareto front for performance, reliability, and costs. Figure 11 shows the response time in seconds over the costs per candidate, while Figure 12 shows the probability of failure on demand over the costs per candidate. The software architect can use these results to make an informed trade-off decision among the different quality criteria.

In Figures 11 to 12, the 58 Pareto-optimal points are highlighted as thick squared marks. These points are not located at the borders of the candidate sets, as it would be the case for a two-dimensional set. Pareto-optimal points located within the set are superior to others in the third quality criterion not shown in the respective diagram (i.e., reliability in the first case and performance in the second case). For example, the highlighted candidate in Figure 11 has a higher response time and higher costs than some other candidates, but its reliability is superior to these other points.

We describe one of the found Pareto-optimal solutions in more detail. It is highlighted in Fig. 11 and Fig. 12 using circles. The response time of this solution is 1.34 seconds (initial candidate: 2.2 seconds), its POFOD is 0.0526 percent (initial candidate: 0.0605), and its costs are 69.83 units (initial candidate: 98 units). Therefore it is superior to the initial candidate in all quality criteria.

The genome of this solution is: [1.76, 1.19, 1.01, 0.53, 1.02, 2, 1.62, 1.71, DB, WebServer3, DB, server1, server1, server1-a, server3d, server2, server1, server1-a, server3d]. This means that the evolutionary algorithm has increased the processing rates of almost all CPUs. Server1 was formerly fully utilised by the WebServer component, but now can also host the Dispatcher component, because of the increased processing rate. Therefore an additional server for the Dispatcher can be saved lowering the overall costs.

The algorithm has also selected the component WebServer3 instead of WebServer, because of its better performance with only slightly higher costs. Furthermore, the optimization run deemed the DB2 component as too expensive and continued to use the DB component. Two of the ReportingServers have been allocated to the same server, thereby saving the costs for an additional server while still providing adequate performance and reliability.

Our main contribution in this work is to formulate the search problem and allow automatic search, and not the particular chosen optimisation strategy. Thus, we have not compared our optimisation run with random search in this work. Such a comparison can be found at².

To conclude, the optimisation run of our case study was ²palladio-approach.net/PerOpteryx
able to reduce the amount of over-provisioning present in the initially modelled system and presented a solution candidate with improved performance and reliability for lower costs. The software architect does not have to use this solution exactly as produced by the optimization run. However, it is now known that faster processors within the system can help to reduce the amount of needed servers, and that buying the WebServer2 component justifies its higher costs.

6. LIMITATIONS AND FUTURE WORK

Besides inheriting all limitations of the underlying quality prediction techniques (see [2] for performance, [25] for reliability), our approach exhibits the following limitations:

- **No guaranteed optimality**: The approach itself is a best-effort approach and does not guarantee to find the real Pareto-front, i.e. the globally optimal solutions, because metaheuristics are used.

- **Questionable efficiency**: As the evaluation of each candidate solution takes several seconds, the overall approach is considerably time consuming. Here, software architects should run it in parallel to other activities or over night. A distribution of the analyses on a cluster of workstations could lead to significant improvements. It could also be possible to split the optimisation problem in several independent parts that are be solved separately and thus quicker.

- **No regard for uncertainties**: For the results, uncertainty of estimations, uncertainty of the workload, and the resulting risks are not taken into account. Here, sensitivity metrics could be an additional quality criterion.

- **Limited degrees of freedom**: Currently, design options that offer new degrees of freedom are not yet considered. For example, adding a new server results in further options to configure that server. Such design options could be integrated by formulating the genotype as a tree structure rather than a vector.

- **Simplistic costs model**: The costs model used here is simplistic, as we only wanted to demonstrate the approach. We do not want to devise a new costs estimation technique. However, more sophisticated estimation techniques such as COCOMO II [4] could be integrated.

An important aspect of future work is to combine our approach with subordinate heuristics to make use of performance domain knowledge. For example, heuristics to improve allocation based on the resource demands of components and utilisation of servers could be introduced. In addition, rules as presented in [38, 11, 28] could be integrated in specialised mutation operators. Performance antipattern detection and solution as suggested in [10] could complement this approach. Such subordinate heuristics can help the approach to find good solutions more quickly.

7. CONCLUSIONS

This paper presents an approach to automatically improve software architectures with respect to performance, reliability, and costs. Using this approach, the design space spanned by different design options (e.g. available components and configuration options) can be systematically explored using metaheuristic search techniques. Based on an initial architectural model of a system, new candidates are automatically generated and evaluated for the quality criteria. The process is extensible for different modelling notations, analysis methods, and quality criteria. We provide a prototypical implementation of our approach with the PerOpteryx tool based on the PCM as well as a case study to demonstrate the feasibility and the benefits.

Our work saves software architects effort to manually explore the design space of their software architecture under study, and instead allows them to focus on Pareto-optimal solutions for their trade-off decisions between multiple quality criteria. In addition, software architecture alternatives can be found that might have escaped human attention. Thus, our approach can reduce development costs and result in software architectures with better quality properties. The next steps to extend our work will be the integration of subordinate heuristics to include performance domain knowledge. Here, we plan to integrate our metaheuristic approach with existing rule-based approaches to leverage the advantages of both. Furthermore, we will enhance our approach to allow the specification and checking of quality criteria requirements as well as manually specified constraints on the architectural model, which can help avoid undesired or infeasible design alternatives.

8. REFERENCES


[9] C. A. C. Coello. A comprehensive survey of


