Institute of Software Technology
Reliable Software Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 335

Handling Quality Trade-Offs in Architecture-based Performance Optimization

Sebastian Frank

Course of Study:	Softwaretechnik
Examiner:	Dr.-Ing. André van Hoorn (Prof.-Vertr.)
Supervisor:	Dr.-Ing. André van Hoorn
Commenced:	Mai 27, 2016
Completed:	November 26, 2016
CR-Classification:	D.2.11 / D.2.8 / I.6.4

Abstract

The goal of software architecture optimization is to find architecture candidates that satisfy the expectations of all relevant stakeholders with regard to some quality attributes, e.g., performance, modifiability, or reliability. Quality attributes usually compete with each other, which makes trade-offs inevitable.

In the SQuAT project the suitability of distributed search strategies for architecture optimization is investigated. This approach is based on the way human architects would conduct architecture optimization. Related works from the domain of software architecture optimization are usually monolithic and only extendable to a certain degree. The SQuAT approach tries to overcome these drawbacks.

This thesis contributes a SQuAT module for the analysis and optimization of software architecture with regard to the quality attribute performance. Therefore, the already existing approach for architecture-based performance optimization PerOpteryx is integrated. In addition, this module gets evaluated with an example system and it is shown that it reaches a similar quality than PerOpteryx. In conclusion, this work is the first step to gain new insights into the applicability of distributed search strategies, modularization of design knowledge, and negotiation techniques for software architecture optimization.

Kurzfassung

Das Ziel von Software Architektur Optimierung ist das Auffinden von Architekturkandidaten, welche die Anforderungen aller relevanten Interessengruppen bezüglich bestimmter Qualitätsattribute, wie z. B. Performanz, Änderbarkeit oder Zuverlässigkeit, erfüllen. Qualitätsattribute neigen jedoch gewöhnlich dazu, miteinander zu konkurrieren, wodurch Kompromisse unausweichlich sind.

Im SQuAT-Projekt wird untersucht, inwieweit verteilte Suchstrategien für die Architektur Optimierung geeignet sind. Dieser Ansatz orientiert sich dabei an der Arbeitsweise von Menschen. Existierende Arbeiten aus dem Bereich der Software Architekturoptimierung sind in der Regel monolithisch und nur begrenzt erweiterbar. Mit dem SQuAT Projekt sollen diese Nachteile überwunden werden.

Diese Bachelorarbeit steuert ein SQuAT-Modul für die Analyse und Optimierung von Softwarearchitektur bezüglich des Qualitätsattributs Performanz bei. Dafür wird der bereits existierende Ansatz für architekturbasierte Performanzoptimierung PerOpteryx eingebunden. Das entwickelte Modul wird außerdem anhand eines Beispielmodels evaluiert und gezeigt, dass es eine ähnliche Qualität wie PerOpteryx erreicht. Diese Arbeit stellt somit die Basis für weitere Erkenntnisse über die Anwendbarkeit von verteilten Suchstrategien, Modularisierung von Entwurfswissen und Verhandlungstaktiken für Softwarearchitekturoptimierung aus dem SQuAT-Projekt dar.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The importance of software quality and architecture optimization, as well as the challenges in architecture optimization are presented in Section 1.1 of this chapter. In Section 1.2 related work to this thesis is mentioned and common restrictions of these approaches are described. The SQuAT approach is presented in Section 1.3 as a new approach to overcome these restrictions and because this thesis contributes to this project. Section 1.4 then summarizes the goals and contributions of this project. Finally, the structure of this thesis is presented in Section 1.5.

1.1. Motivation

The main objective of architecture optimization is to fulfill the requirements of a software system's stakeholders regarding different quality attributes. This is done by evaluating different variants of software architectures without changing the functionality of the software system. Quality attributes like performance, reliability, or modifiability are influenced by architectural changes.

While software developers have laid there focus mainly on the functional requirements in the past, many quality attributes are today considered important for the success of software projects. For some software projects over 90% of the total costs are caused by the maintenance and the management of the software system's evolution [Erl00]. In addition, approximately 50% of the working time of software maintainers is spent on understanding the code, which has to be maintained [FH83] [Sta84]. Improvements to the modifiability of software systems can thus be a sufficient measure for software companies to save a lot of money. Other quality attributes, like performance, can highly influence the satisfaction of the users. Google conducted several experiments on their services [Far06]. In one experiment they increased the results per page from 10 to 30.

While the loading time was only half a second longer, the searches decreased by 25%. This shows how much performance is related to user satisfaction, especially for web applications.

Reducing costs and increasing the satisfaction of their customers should always be the goal of every software company to be successful. Nevertheless, satisfying quality attributes is most times not an easy task for big software systems. Some software systems today can easily have millions of SLOCs, e.g., the source code of Red Hat Linux 7.1 contains over 30 million physical SLOC [Whe]. The size of such systems offers many possibilities for combinations of changes to influence quality attributes. Besides, the environment of the software can be changed, too. Even if the optimization is limited to only one quality attribute, this task could be a lot of work for a human software architect.

Considering more than one quality attribute leads to even more challenges. Many quality attributes tend to compete with each other, because changes to the software architecture can have side effects on other quality attributes, e.g., adding additional abstract layers usually increases modifiability, but obviously causes the performance to decrease. As a result there is usually not only one best solution, there are several Pareto-optimal solutions. For that reason it is necessary to make trade-offs. At some point the architect must determine his preferences and then choose the architecture, which fits them the best. Therefore human participation in the optimization process is considered useful, but the optimization itself should be done by tools due to the huge size of the design space.

1.2. Limitations of Existing Works

The size of the design space is usually so big, that even the most recent approaches are based on heuristics. Nevertheless several approaches exist, they are often limited to one or a few quality attributes from a small group of well discovered quality attributes, e.g., performance, reliability, and modifiability. This restriction is also related to the assumption of an underlying architectural model, which is usually not applicable for other quality attributes. Furthermore, the knowledge needed for the optimization is often hard-wired within the tool and highly coupled with the underlying architecture model, which makes these tools hardly extendable. Some tools already offer humans the possibility to influence search, however most of them are far from being interactive.

Many rule-based systems for performance optimization exist. One of them is Performance Booster [Xu12], which is based on Layered Queueing Networks (LQN). However, the optimization capabilities of these approaches are limited by the availability of rules.

Other approaches, like Planner2 [ZW03] and CERAS Deployment Optimization [Li+09] are specialized on specific tasks. Planner2 is also based on Layered Queueing Networks (LQN) and changes scheduling and allocation. CERAS Deployment Optimization minimizes cost with regard to performance constraints. However, these approaches are too specialized to use them for other tasks or to extend them for other quality attributes.

Some more general applicable and extendable approaches exist as well, e.g., ArcheOpterix [Ale+09], SASSY [MCD08], GDSE [SK10], and ArchE [Bac+05]. However, ArcheOpterix is restricted to the Architecture Analysis & Design Language (AADL) and focuses on embedded systems. SASSY is only extendable with regard to the limitations of service-oriented systems and quality aggregation functions, which are used in this approach. The quality attributes of GDSE can also be extended, but only as far as they are expressible by an arithmetic function. ArchE is different from these approaches, because the knowledge for each quality attribute is provided by separate frameworks. However, this approach is more an assistant for architecture optimization and automated strategies for negotiation are missing.

1.3. SQuAT Approach

The research project SQuAT (Search Techniques for Managing Quality-Attribute Trade-offs in Software Design Optimizations) conducted by the University of Stuttgart and the Universidad Nacional del Centro (UNICEN) tries to overcome these drawbacks by investigating a new semi-automated approach for design space exploration by particularly focusing on three major points. Firstly, distributed search techniques are investigated for this project. Secondly, the approach should reach a high modularization of design knowledge. Finally, negotiation techniques for managing quality attribute trade-offs should be applied.

Instead of using a monolithic analysis, this approach is based on local optimization and negotiation, inspired by how human architects would do architecture optimization in real life. A human architecture analysis team would presumably be composed of experts for each quality attribute, the client, and a product manager. The experts contribute the knowledge to analyze and optimize the architecture for their particular quality attributes, e.g., performance, modifiability, and reliability. To achieve a satisfying result, it is necessary for all the people to discuss and negotiate in order to find suitable trade-offs. The participation of the client will make sure that his requirements and priorities will be considered all the time in the negotiation. In contrast to that, the product manager adds additional constraints to the negotiation, especially regarding the development time and the project budget. In the end it is unlikely to have a result that makes everybody happy, but a compromise can usually be achieved.

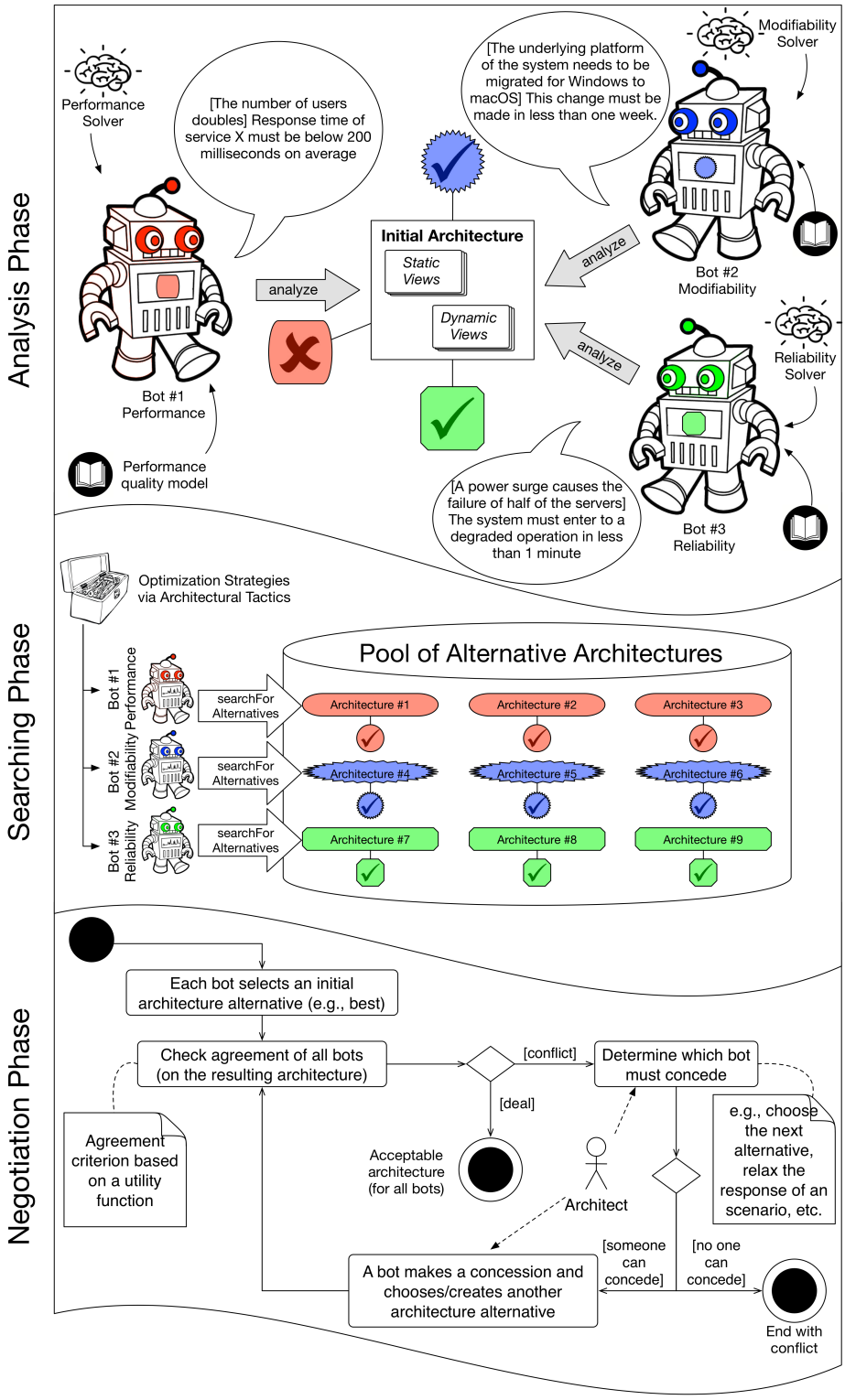

Figure 1.1.: The SQuAT approach consists of the analysis phase, the searching phase, and the negotiation phase. [Pac+16]

Instead of humans, the SQuAT approach uses specialized modules, which are named bots. These bots participate in three different phases. The first phase is the analysis phase, in which the initial architecture gets analyzed. In the second phase, the bots search for alternative architectures. Finally, in the negotiation phase the results will be compared and it is tried to reach an agreement on one architecture. All three phases are shown in Figure 1.1 in more detail.

The analysis performed by the bots is based on scenario templates as the ones defined by the SEI at Carnegie Mellon, like for instance used in ATAM (Architecture Tradeoff Analysis Method) [CKK02]. In these scenarios a stimulus to an artifact results in a response, which can be measured. These measurements can be compared to the architects expectation and thus the bot is able to determine whether an architecture satisfies these requirements or not. Each instance of a bot processes one scenario. To solve this scenario a bot must contain its own solver and architectural model. The input model itself is a more abstract and general model, which will usually be transformed to a more specific model by the bot.

In the searching phase every bot performs a local optimization, to find alternative architectures. Therefore each bot is able to apply tactics, which can be described as transformations of the architecture to improve the satisfaction of a single quality attribute. Performance tactics often perform changes on the hardware or used resources, whereas modifiability tactics are often changes in the modularization of the software. Nevertheless tactics can be quite different from each other. Every alternative architecture can be described as a number of tactics applied to the initial architecture. Every instance of a bot will try to return a certain number of alternative architectures at the end of the phase, which at least satisfy their own scenario. The optimization techniques used by each bot can be different for each bot. Thus, bots can make use of optimization techniques, which usually deliver good results for their quality attributes, e.g., in the domain of performance, evolutionary algorithms are considered to be a good choice.

The bots have to choose one of their found alternative solutions as their favorite candidate, before the negotiation phase begins. The process is very simple as long as at least one of these solutions satisfies all other bots. If this is not the case, one bot has to concede, either by choosing another alternative solution, or by decreasing the expected response measure of its scenario. At some point, both options might not be applicable, then the negotiation will end with a conflict, causing the whole process to fail. The negotiation phase also offers some suitable situations for human architects to interactively influence the outcome of the negotiation. The architect knows the priorities for each quality attribute and can therefore assist in the decision which bot and how a bot has to concede.

The SQuAT approach is currently not yet evaluated and still work in progress. For a first evaluation of the project it is planned to implement a Performance Bot and Modifiability

Bot, both based on the Palladio Component Model. While the University of Stuttgart is responsible for delivering the Performance Bot, the UNICEN researchers implement the Modifiability Bot. If the suitability of this approach can be shown, it is planned to extend it in later steps, e.g., by a more abstract architectural input model and more bots.

1.4. Goals

For this thesis the main goal is to develop an approach for performance architecture optimization, that conforms to the bot interface of the SQuAT project. This approach should be able to analyze and optimize an initial Palladio Component Model [BKR09]. It should return not only the response measure for common performance metrics, but also the changed models of the investigated candidates. Therefore the approach should contain its own solver, optimization technique and underlying architectural model. It is also necessary to evaluate the applicability of this approach by comparing it to another state-of-the-art approach.

While this thesis contributes a new approach to the domain of performance architecture optimization, it is also the precondition to foster new insights into architecture optimization in general by the SQuAT project. This will lead to a better understanding of the applicability of distributed search techniques for architecture optimization. Besides, the resulting approach will be more extendable than existing approaches and one of the first approaches that considers negotiation for architecture optimization. In addition, the evaluation can be reused and extended for this project.

1.5. Thesis Structure

This thesis is structured in the following way:

Chapter 2 – Foundations: This chapter contains the theoretical knowledge necessary to understand how software architectures can be evaluated and optimized. In addition, the technologies used for the Performance Bot are described in more detail.

Chapter 3 – Performance Bot: The developed approach for the Performance Bot is described here.

Chapter 4 – Evaluation: An example model and the evaluation of the approach is presented in this chapter.

Chapter 5 – Conclusion: Concludes the results of this work and mentions possible future works.

Chapter 2

Foundations

This chapter gives an overview of the foundations of architecture optimization. Although software architecture optimization is not restricted to a single domain, this chapter partially focuses on architecture optimization from the perspective of performance, because the goal of this work is to provide a bot for SQuAT, which contains the knowledge for performance architecture optimization. However, there is also information about other quality attributes presented in this chapter to provide a better understanding of architecture optimization in general.

Section 2.1 presents different quality attributes which are usually considered for software architecture optimization. Section 2.2 describes how these quality attributes can be measured through metrics. In Section 2.3 typical transformations on the software architecture are outlined. As architecture optimization is performed on models, an overview of different models is given in Section 2.4. Section 2.5 introduces the concept of scenarios to evaluate the satisfaction of non-functional requirements, while Section 2.6 presents methods to evaluate software architectures in practice.

Section 2.7 introduces the general optimization process for software architecture optimization. Methods for multi-criteria optimization are subsequently outlined in Section 2.8. Following this, different approaches and strategies for software architecture optimization are summarized in Section 2.9. The most frequently used optimization strategy, evolutionary algorithms, is subsequently presented in Section 2.10. Section 2.11 outlines possible types of optimization goals.

The most important technologies used in this thesis are described at the end of this chapter in more detail. Section 2.12 outlines the characteristics of the Palladio Component Model, which can be used to model software architectures. The Palladio-Bench supports architects with modeling and evaluation of instances of the Palladio Component Model. Therefore, it is presented in Section 2.13. Section 2.14 presents PerOpteryx, a

multi-objective software architecture optimization approach, which is also integrated into the Performance Bot described in Chapter 3.

2.1. Quality Attributes

As the goal of architecture optimization is to improve the quality of software without changing the functionality, quality attributes must be used to describe this quality. In general, the relevant quality attributes can be derived from the non-functional requirements of the software. While the selection of relevant quality attributes can vary between software systems, some of them appear more often than others.

Quality Attribute	Percentage
Performance	44%
Cost	39%
Reliability	37%
Availability	13%
General	12%
Energy	9%
Weight	3%
Safety	2%
Reputation	2%
Modifiability	2%
Area	2%
Security	<1%

Table 2.1.: Percentage of papers investigated by Aleti et al. [Ale+13] which optimized the mentioned quality attribute.

The literature review of Aleti et al. [Ale+13] investigated 188 research papers from the domain of software architecture optimization. Among other things, it was investigated which quality attributes have been considered in these papers. The results, which are shown in Table 2.1, indicate the importance of the three most commonly used quality attributes performance, cost, and reliability. They are optimized in about one third of the investigated papers. Therefore, they can be considered as well investigated and important in practice. However, there are also quality attributes which can not be considered as classical software quality attributes, e.g., energy consumption or weight. Other quality attributes, e.g., security, are considered as important in practice [BCK03], but are not well investigated. This indicates that automatic optimization of some quality

attributes is more easy than for others. Some quality attributes also appear only in specific domains, e.g., safety in embedded systems.

Cost is obviously the most important quality attribute for most companies, because the cost of a software system should not exceed its monetary benefit. At some point the improvement of other quality attributes will influence the cost of the software system. As a result, considering costs together with other quality attributes, will usually lead to trade-offs.

2.2. Quality Metrics

It is necessary to quantify quality attributes in order to compare software architectures. Quality metrics can be used for this task. Typical performance metrics are response time, throughput, and resource utilization [Jai90]. Response time can be measured by the time difference between sending a request to the software system and receiving the answer. Throughput describes the number of tasks, that can be performed within a time span. Resource utilization on the other hand describes how long a resource, e.g., the CPU, is utilized within a time span.

Other quality attributes have to be measured in a different way. Reliability, for example, states whether a software system provides its functionality as expected. Therefore, a commonly used reliability metric is the probability of failure on demand (POFOD) [Koz14]. For cost, often a common currency is chosen, e.g., Dollars [Koz14]. It is also possible to compute the cost of a software system as the sum of its components development costs. However, the costs for the whole software life cycle can be considered. In addition to the development costs, there are often maintenance costs, hardware procurement costs, operating costs and licensing costs.

For some quality attributes, it is harder to find suitable metrics. One approach uses the number of dependencies as metric for the modifiability of software systems [YC88]. The underlying assumption is that a high intramodular and intermodular localization also increases the modifiability of the software system.

2.3. Degrees of Freedom

In architecture optimization, changes to the architecture lead to a change in the quality of the software system. It is therefore necessary to have degrees of freedom in the architecture which influence the specified quality attributes. The literature review of Aleti et al. [Ale+13] also investigates which degrees of freedom are commonly used.

Table 2.2 presents the results of the investigation. The most common degree of freedom is allocation. Software components can be allocated to different servers to influence the quality of the software system. Other frequently used degrees of freedom are hardware and software replication. While software replication is restricted to changes in the number of copies of software entities, hardware replication is related to redundancy of hardware, e.g., a second database sever is provided.

Degree of Freedom	Percentage
Allocation	31%
Hardware replication	21%
Hardware selection	20%
Software replication	18%
Scheduling	17%
Component selection	16%
Service selection	15%
Software selection	13%
Other problem specific	9%
Service composition	6%
Software parameters	5%
Clustering	3%
General	3%
Hardware parameters	2%
Architectural pattern	2%
Not presented	2%
Partitioning	1%
Maintenance schedules	1%

Table 2.2.: Percentage of papers investigated by Aleti et al. [Ale+13] which used the mentioned degree of freedom.

Not all degrees of freedom have the same impact on every quality attributes. While scheduling is likely to influence the performance of the software system, it will probably have no effect on modifiability. For that reason, quality attributes usually appear together with only some of the existing degrees of freedom. Allocation, scheduling, and service selection are frequently used to influence performance, for instance.

Most degrees of freedom together with different quality attributes. That is also the reason for trade-offs being necessary in software architecture optimization. For instance, hardware selection can influence the three quality attributes performance, reliability, and cost at the same time.

2.4. Models

The purpose of software architecture optimization is to choose the best architecture for a software system. Usually the architecture of a software system gets already defined in the design phase of a software project. As a consequence, it is often not possible to get measurements for the real system. Because of that, software architecture optimization is usually applied on models, because they can be created before the implementation. However, it is also possible to use measurements from an existing software systems to improve the model (also see Section 2.6).

Aleti et al. [Ale+13] describe three general categories for models in software architecture optimization. The first category is named "architecture models" and summarizes models based on components and connectors to describe the architecture of the software system. The Unified Modeling Language (UML) [RJB04] is a popular example of this category. Other architecture models are the Architecture Analysis and Design Language (AADL) [Aad] and the Palladio Component Model (PCM) [BKR09]. These two models were both developed for the purpose of architectures analysis. "Evaluation models" do not model the architecture, but can be used to evaluate the software architecture, e.g., Markov Chains [Nor98] and Layered Queueing Networks (LQN) [Fra+09]. As third category the "optimization models" are described. These models connect the architecture and the design decisions to the decision variables and the objective function. They are used in every optimization process, but they can be derived from models of the other two categories.

It should be kept in mind that models are only abstract representations of the reality. By modeling a software system some information could get lost, either because the model is not capable of storing this information, or because the modeler decides not to model everything. LQNs for example represent a view on the hardware of the system, whereas UML is designed for the representation of software. The UML MARTE [(OM] profile however extends the basic UML to be applicable for performance modeling.

If two models are in some way similar to each other, then it is also possible to apply transformations, e.g., for PCM to LQN [KR08]. Properties which can only be represented in one of these two models, will get lost in this transformation. In conclusion, the choice of the initial model is important in architecture optimization, because it decides which degrees of freedoms are available in the optimization process. Therefore, it is also a decision about which quality attributes can be considered in the approach.

2.5. Scenarios

Scenarios [CKK02] are a concept to investigate, whether non-functional requirements are satisfied or not. In the SQuAT approach, many scenarios can be defined and each bot is responsible for exactly one of these scenarios.

A scenario consists of six parts: the source, the stimulus, the environment, the artifact, the response, and the response measure. The source is an actor which generates the stimulus, e.g., a human or a computer. The stimulus is best described as a condition that arrives at the artifact, which can be the whole system or pieces of it. This can also happen under certain conditions which are described by the environment. The reaction to the stimulus is the response. A response can be measured with a quality metric. Finally, it is possible to compare this response to the expectation and decide whether the requirement is fulfilled or not.

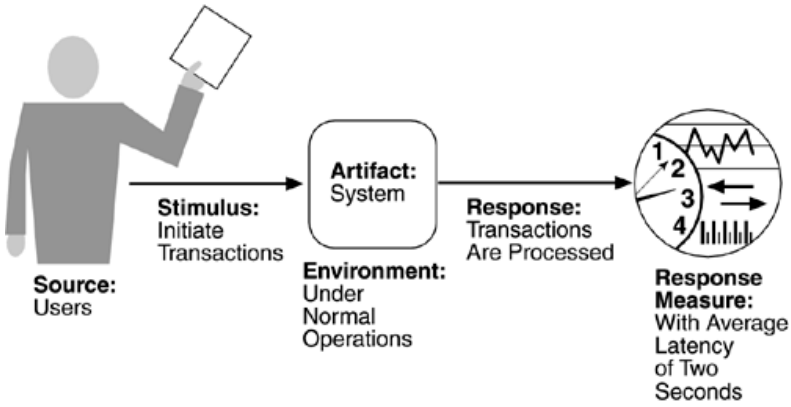

Figure 2.1.: Sample performance scenario. [CKK02]

Figure 2.1 illustrates the performance scenario “Users initiate 1,000 transactions per minute stochastically under normal operations, and these transactions are processed with an average latency of two seconds”. This example uses the response time as metric for performance. It is typical for performance, because performance scenarios describe usage situations, while modifiability scenarios are usually related to a desired change of functionality.

2.6. Architecture Evaluation

Depending on the current development phase of the investigated software system, there are different methods to evaluate this system. For a running system it is possible to measure the responses for the system for a real stimulus. Some approaches already

exist, which can monitor software systems, e.g., Dynatrace [Dyn], AppDynamics [App], and Kieker [VHWH12]. These measurements can be event-driven or sampling-based. In any case, it has to be considered that every measurement method also influences the measured system. Nevertheless, models created on the base of measurements often achieve a higher accuracy than without measurements [Bru+15]. It is even possible to use model generators to model software systems based on measurements, e.g., like in the approach of Willnecker and Krcmar [WK16].

Measurements can at least be used for the initial candidate for software architecture optimization, but in most cases it is impracticable to implement every generated architecture to evaluate it. In software architecture optimization two different types of technologies exist for the evaluation of model instances. Solvers calculate the metrics based on mathematical methods. They are able to treat the model as equation, which has to be solved. Solvers exist for some models, e.g., a solver for LQN [Fra+09]. In contrast to solvers, simulators are able to operate on the model itself. They simulate the model and measure the behavior of the simulated system. EventSim [MH11], for instance, is an existing approach for PCM. Simulators are usually considered to be more accurate, but slower than solvers.

2.7. Optimization Process

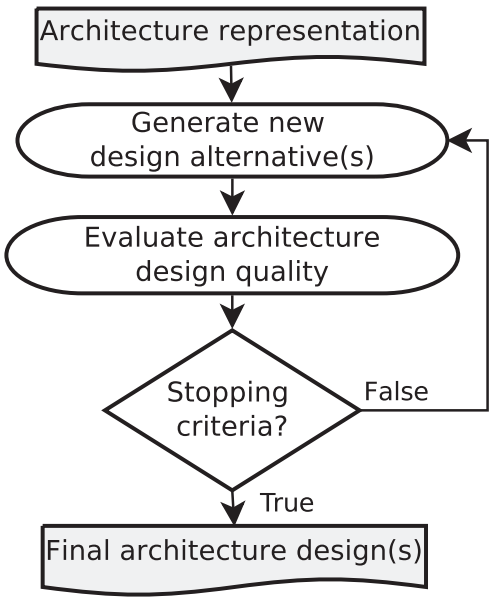

Figure 2.2.: General optimization process [Ale+13].

The general optimization process for software architecture optimization is shown in Figure 2.2, as described by Aleti et al. [Ale+13]. As input for the optimization process a representation of the architecture has to be provided. This is a model from one of the three categories described in Section 2.4. While human readable input models are often preferred, performing transformations internally are common practice to achieve a more suitable presentation for the optimization. Depending on the chosen optimization strategy, at least one alternative architecture is generated. To enable a comparison between the different architectures, the quality of these architectures have to be evaluated. Solvers or simulations are often used in this step, as described in Section 2.6. A stopping criteria must be implemented to decide, whether more candidates have to be generated. Ff the optimization is successful, the output of the optimization process will be the improved architecture for the software system.

2.8. Multi-Criteria Optimization

A real software system can have several non-functional requirements for different quality attributes. Thus, architecture optimization also has to be applicable for multiple scenarios. Each objective can then by seen as one dimension forming a multi-dimensional design space. Each candidate represents a vector in this design space after its evaluation. Three different methods can exist to deal with this situation [Koz14].

The “a priori method” suggests to capture the preferences of the stakeholders to generate a preference model. As a result, a scalar objective function can be defined and all solutions are ranked. Therefore, this method transforms a multi-objective optimization problem into a single-objective optimization problem. On the one hand this problem is easier to solve, on the other hand it is usually difficult for the stakeholders to define realistic preferences for the quality attributes.

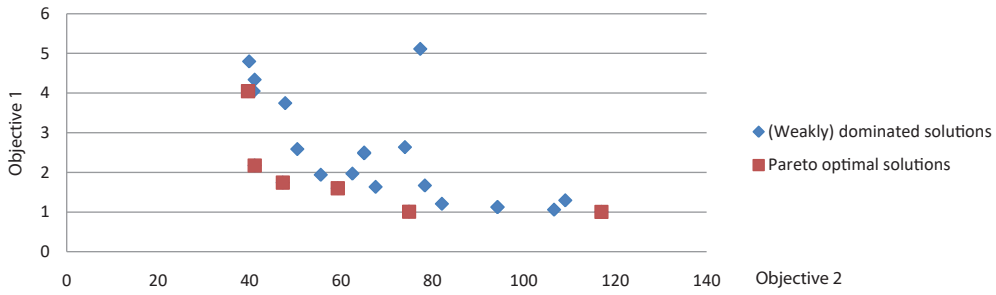

Figure 2.3.: Example for Pareto Optimal Solutions [Koz14].

The “a posteriori method” moves this decision behind the optimization process. In this method several solutions are presented instead of a single best solution. The concept of

Pareto-optimality can be used to determine a group of best solutions. Pareto-optimal solutions can not be compared to each other without defining a preference model. For example, it is not possible to generally say a solution with a high modifiability and a low performance is better than a solution with low modifiability and high performance. However, there is always at least one Pareto-optimal solution for every non-Pareto-optimal solution that achieves better values for all objectives. Figure 2.3 illustrates this concept for an example with two objectives.

Finally, the "interactive method" enables the participation of a human architect in the optimization process. The tool presents intermediate solutions to the architect, who can then influence the further optimization by adjusting the preference model. In general, a priori methods can always be extended to become interactive methods.

2.9. Optimization Strategies

Due to the size of the design space, the majority of the software architecture optimization approaches uses strategies which provide only approximate solutions. Nevertheless, some approaches use optimization methods like standard mixed integer linear programming or problem-specific methods to find the exact solution of the optimization problem [Ale+13].

For performance, rule-based approaches, specialized approaches, and metaheuristic approaches can be distinguished [DG+12]. Rule-based approaches, like Performance Booster [Xu12] and ArchE [Bac+05], use domain-specific knowledge to optimize a software architecture. The rules defined for these tools are often based on experience and limited to the domain of a particular quality attribute. Although specialized approaches are more efficient, they are limited to performance and certain tasks, e.g., the CERAS Deployment Optimization [Li+09] to deployment.

Metaheuristic approaches are almost independent from domain-specific knowledge. Therefore, they are not limited to a particular quality attribute. ArcheOpterix [Ale+09] and SASSY [MCD08], for instance, use metaheuristic approaches. Two different types of metaheuristics can be distinguished, namely trajectory methods and population-based methods [BR03]. Trajectory methods perform a local search around a single candidate in order to find a better one. Because of that, they are difficult to apply for multi-objective methods. Simulated annealing, tabu search, and variable neighborhood search [BR03] are commonly used trajectory methods. Population-based approaches generate multiple solutions in one iteration. For this reason they can be easily applied to multi-objective optimization problems. Besides evolutionary algorithms (see Section 2.10), ant colony

optimization and swarm optimizations [BR03] are also categorized as population-based approaches.

2.10. Evolutionary Algorithms

Evolutionary algorithms [CLV06] are a common way to solve multi-objective problems for in the domain of software architecture optimization. They are based on the concept of evolution from biology. Figure 2.4 presents the basic evolutionary algorithm, which consists of three repeatedly executed phases. The input for the algorithm are n randomly generated candidates, in which n is the size of the population. For software architecture optimization, one of these candidates is the initial candidate which should be optimized.

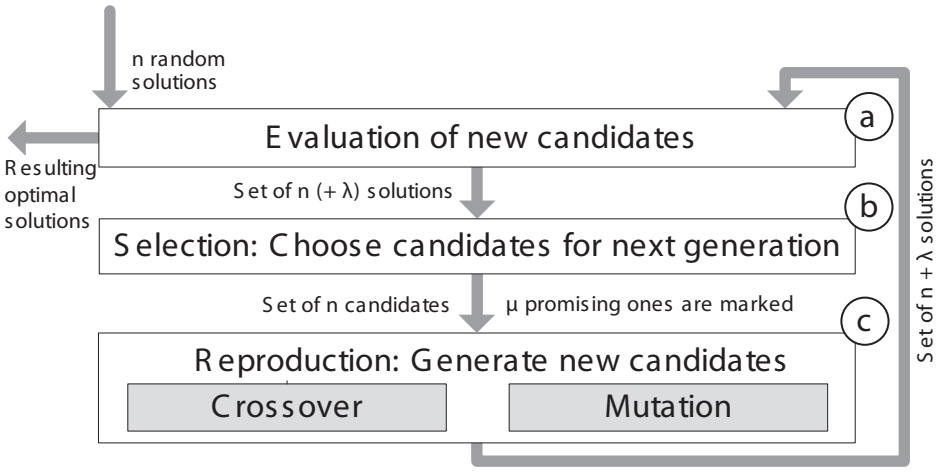

Figure 2.4.: Basic evolutionary algorithm [Koz14].

In the evaluation phase (a), all not already evaluated candidates get evaluated. After this phase the candidates can be ranked and the best candidates can be determined. For the evaluation, the objectives can be weighted or an approach based on the Pareto-dominance is used, e.g., the Pareto rank or Pareto strength. To rank the candidates within a group of Pareto-optimal solutions, density measures can be used to assure a high diversity of solutions [Deb+02].

The ranked solutions can now be filtered in the selection phase (b). Only the n best solutions will stay part of the population after this phase. Of course, no candidates are removed in the first iteration, because there are only n solutions at this point. In addition, a group of μ candidates is selected as parents for the next step in the algorithm.

A common selection strategy is the tournament selection. In the tournament selection a candidate with better ranking has a higher probability to be chosen.

The last step is the reproduction phase (c), in which λ new candidates are generated. Usually two different operations are applied on the chosen parents in this phase. The crossover operation combines parts of two candidates to generate a new candidate. A fixed cut point, a randomly chosen cut point, or a random selection for each degree of freedom can be used to select these parts. While the crossover operation tries to improve candidates by combining two promising candidates, the mutation operation applies small random changes to a single candidate. Again, there are different possible configurations for this operation. The number of degrees of freedom to change can be random or static. Finally, the whole population is passed back to the evaluation phase.

If the stop criterion is satisfied, the algorithm returns the optimized candidates. For multi-objective optimization, this is a set of Pareto-optimal solutions. Bounded archives can be used to store the current Pareto-optimal solutions to assure that no already investigated Pareto-optimal candidate is removed in the selection phase. Evolutionary algorithms which always return all Pareto-optimal candidates are called elitist.

Heuristics offer the possibility to further improve evolutionary algorithms. Other reproduction operators can be integrated into the reproduction phase. PerOpteryx [Koz14], for instance, integrates so-called tactics for performance and cost, which contain domain-specific knowledge. It has been shown, that these tactics can speed up the evolutionary algorithm [KKR11].

While the starting populations for evolutionary algorithms are usually generated randomly, it is also possible to use starting population heuristics [Gre87]. For example, a starting population created by a simplified quality prediction and limited degrees of freedom has been shown to be beneficial in a case study [Mar+10].

2.11. Optimization Goal

Koziolek [Koz14] presents different ways architects can formulate the goal of the optimization. In any case, a critical value must be defined based on the requirements. If this value is reached, the requirement is satisfied and no further optimization is necessary. It is also possible to define a quality bound instead that should be optimized further. The optimization will stop, if all requirements are fulfilled.

Additional stopping criteria should also be formulated for evolutionary algorithms. Sometimes not all quality criteria can be fulfilled. Thus, the maximum number of iterations or a time limit should be set. It is also possible to use more sophisticated

stopping criteria, e.g., the significance of the Pareto-front improvements after a number of iterations.

2.12. Palladio Component Model

The Palladio Component Model (PCM) [BKR09] is an architecture modeling language designed for component-based software architectures (CBSA). An instance of PCM consists of different models, which can be provided by different persons. PCM allows persons with four different roles to contribute to a PCM instance, because huge software systems are usually created by teams and not every person knows everything about the whole system.

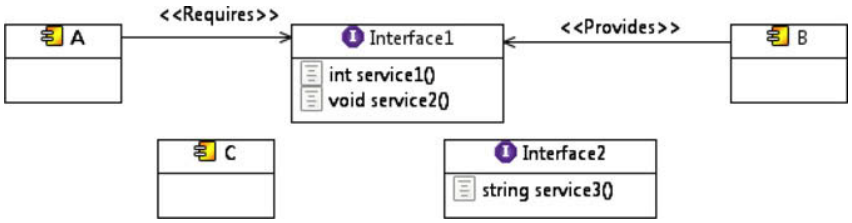

Figure 2.5.: Repository example. [BKR09]

Component developers are responsible for the development of software components and are able to describe which interfaces a component provides and requires. Each of these components can be added to a repository model, as shown in a simple example in Figure 2.5. Additionally, they are also responsible for the implementation of the services of the components. Therefore, they can describe the control flow of these services in an abstract way. Service effect specifications (SEFF) represent the control flow as internal and external actions. While internal actions describe resource demands on hardware, external actions represent calls to other components within the software system. Resource demanding service effect specifications (RDSEFF) can be used to annotate the components with performance annotations, e.g., number of CPU instructions. An example for a RDSEFF, also containing a branch and a loop, is shown in Figure 2.6.

If components have been added to the repository model, the software architects can assemble the components. The system is modeled based on assemblies and interfaces in an architecture model. Existing components can then be linked to the assemblies with regard to their interfaces. Figure 2.7 shows an example for such a system model, in which component *A* is used twice.

System deployers are the experts for the environment of the software system. Because of that, they model the available hardware resources, e.g., servers and links between the

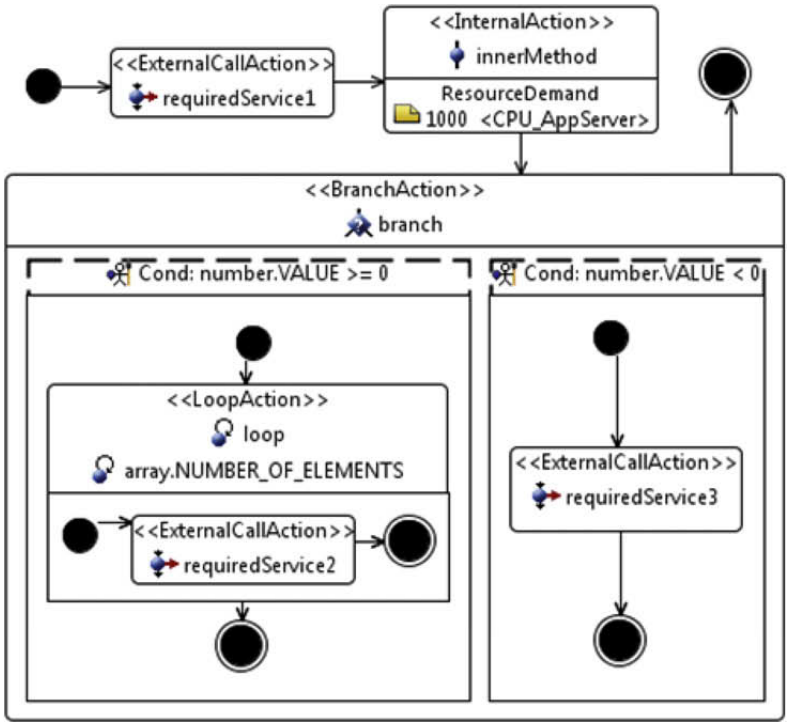

Figure 2.6.: RDSEFF example. [BKR09]

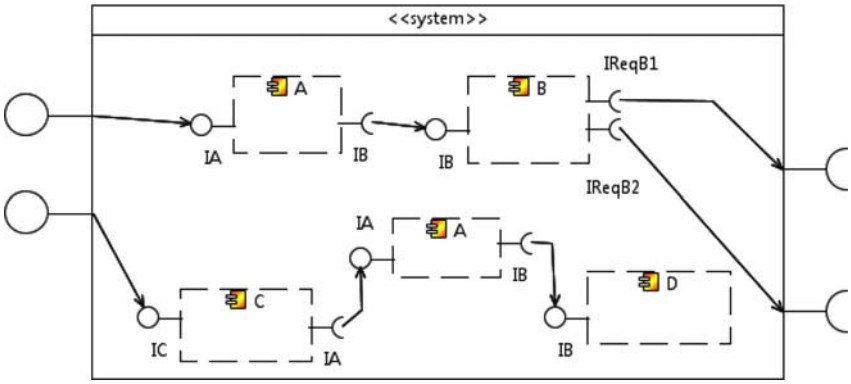

Figure 2.7.: System exmaple. [BKR09]

servers. Another task for system deployers is to annotate the models with the hardware specifications, for example, processing rates of the CPU and the HDD. It is also possible to annotate the scheduling policy and reliability properties, e.g., mean time to failure (MTTF) and mean time to repair (MTTR). System deployers are also responsible for modeling the allocation of the components to the hardware resources.

Business domain experts can contribute a usage model to the PCM instance. Usage models describe the interaction of humans with the software system. Open or closed workloads can be defined and their properties can be selected. In addition, the business

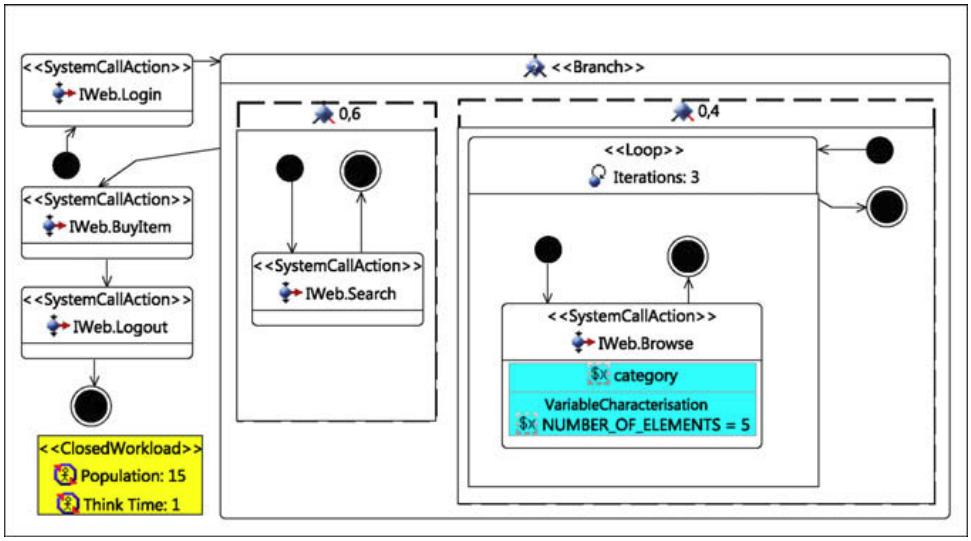

Figure 2.8.: Usage model example. [BKR09]

domain experts can model the usage of the system. They describe which methods are called and how often. A complete flow of calls can be created with branches and loops, as shown in the closed workload example in Figure 2.8.

Other features of PCM are that also failure probabilites of software components and software resources can be modeled. In addition, PCM instances can be annotated with fixed and variable costs for software components and hardware. In conclusion, a PCM instance can contain several degrees of freedom for architecture optimization. Software and hardware as well are modeled, which makes PCM applicable for quality evaluation of different quality attributes.

2.13. Palladio-Bench

The Palladio-Bench is the integrated modeling environment for PCM. This project is open source and integrated into the Eclipse IDE [Ecl]. The Palladio-Bench supports the software architects by providing a graphical editor for the creation of PCM instances. This editor uses an UML-like concrete syntax, as shown in the figures from Section 2.12, to simplify working with PCM for most users.

However, the Palladio-Bench is not limited to modeling, it also allows to evaluate PCM instances. PCM is designed to be unambiguous. Therefore, some approaches for model-to-model and model-to-code transformations can be applied to it. The transformed model is then solved or simulated to evaluate it for a particular quality attribute. In addition, the Palladio Bench can create a performance prototype for measurements.

It is also possible to create Java code skeletons to start with a real implementation of the system. Thus, the Palladio-Bench supports methods to evaluate PCM instances in different software development phases. The basic concept of the Palladio-Bench is also illustrated in Figure 2.9.

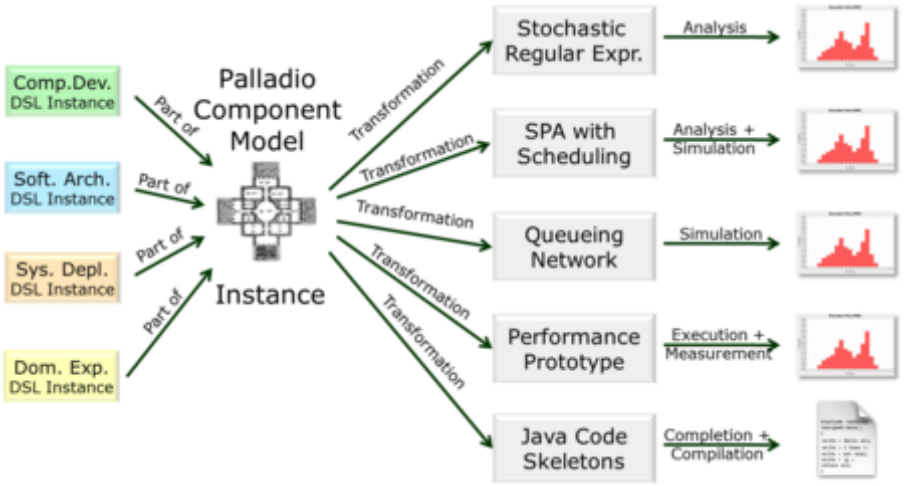

Figure 2.9.: Concept of the Palladio Bench. [Pal]

For performance, transformations to different types of queueing networks can be applied. The discrete-event simulator SimuCom [BKR09] applies a model-to-code transformation to evaluate PCM instances for response time, throughput, and resource utilization. A PCM-to-LQN transformation [KR08] can be applied, which enables the use of simulation or solving. The LQN solver [Fra+09] applies a heuristic performance analysis. Although it is usually fast, it does not support all PCM features, e.g., passive resources. While the original Palladio-Bench is integrated into Eclipse, a Headless LQN Solver [Kel16] has also been developed. This approach runs Palladio with the LQN Solver without Eclipse, because of that it is called “headless”.

The Palladio-Bench is not limited to performance, it also supports the evaluation of reliability, cost, and maintainability. The evaluation methods are different for each quality attribute, for instance, reliability evaluation can be performed by a solver for Markov chains. Another feature of the Palladio-Bench is the visual representation of the evaluation results in graphs. This feature improves the usability of the tool and supports the result interpretation.

2.14. PerOpteryx

PerOpteryx [Koz14] is a plugin for the Palladio-Bench, which is able to perform multi-objective software architecture optimization on an initial PCM instance. This plugin is designed to present Pareto-optimal solutions to software architects in order to support them by making trade-offs. In the categorization presented in Section 2.8, PerOpteryx would be categorized as a posteriori approach. The approach has also been proofed to be applicable for complex software systems in an industrial case study [DG+12].

The quality attributes performance, reliability, and cost can be optimized. In addition, it is possible to define constraints for these quality attributes. Both simulators and the LQN Solver can be used to determine performance values for response time, throughput, and maximum CPU utilization. POFOD can be predicted by the solver based on Markov chains for reliability. A solver for cost is also available, which can determine general cost, initial cost, and operating cost.

Although only performance, reliability, and cost can be chosen by default, the approach is considered to be extendable for other quality attributes due to the used evolutionary algorithm. However, this is only possible with regard to the capabilities of PCM and the available degrees of freedom. Up to six different types of degrees of freedom can be used, namely component selection, component allocation, continuous processing rate, scheduling, passive resource capacity, and server replication. Degrees of freedom in a PCM instance can be detected automatically, so manual configuration is not absolutely necessary.

The implemented optimization strategy is an evolutionary algorithm. Therefore, the maximum number of iterations and the population size has to be defined. The evolutionary algorithm is the NSGA-II [Deb+02] which has been shown to converge faster and assures a high diversity than similar approaches. The evaluation is based on the Pareto rank in combination with a density measure. In addition, the algorithm is elitist.

Performance and cost tactics are implemented to speed up the optimization process. Koziolek [Koz14] describes five tactics for performance and two for costs. “Spread the Load” suggests a reallocation of components or resources, if the load is not evenly distributed. If a high resource utilization is detected, “Scale-Up Bottleneck Resources” suggests to use better hardware. “Scale-Down Idle Resources” is applied for low resource utilization instead and uses cheaper hardware. In addition to this cost tactic, the second cost tactic “Consolidate Servers” can be applied, if no cheaper hardware is available. The tactic “Scale-Out Bottleneck Resources” considers to buy additional hardware, if the existing hardware can not be improved much further. If much communication between components is detected, the performance tactic “Reduce Remote Communication” moves

these components to the same server in order to reduce the communication link utilization. The fifth performance tactic is called “Remove One Lane Bridge” and increases the capacity of passive resources, if their waiting queue exceeds a certain length. All these tactics can be formalized to be strict rules. Some thresholds and tactic weights can be changed in the run configuration of PerOpteryx.

In contrast to the Palladio-Bench, the results of the PerOpteryx are not visually presented to the software architects. Instead, the predicted quality values and the used design decisions are exported to CSV Files. Besides, additional features are available, e.g., stopping criteria and starting population heuristics.

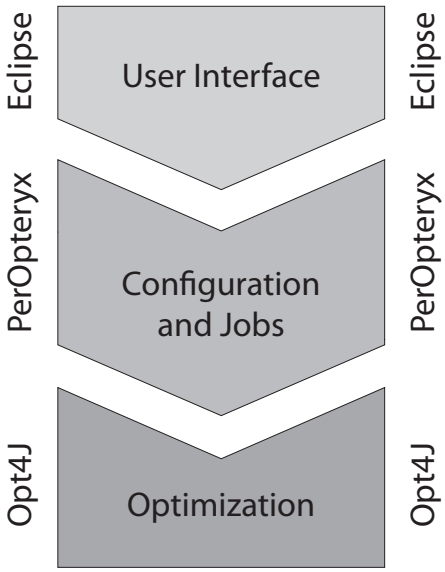

Figure 2.10.: Abstract software architecture of PerOpteryx

The architecture of PerOpteryx is best described as three layer architecture and an abstract presentation is shown in Figure 2.10. As PerOpteryx is a plugin for the Palladio-Bench, and the Palladio-Bench is integrated into Eclipse, the first layer mostly consists of classes which implement interfaces related to the Eclipse IDE. This layer is therefore strongly coupled to Eclipse. The optimization framework Opt4J [Luk+11] is used to realize the optimization itself. PerOpteryx actually connects the other two layers and implements the remaining logic, which is represented as the layer in the middle. However, there are many dependencies between the PerOpteryx layer in the middle and the other two layers.

Chapter 3

Performance Bot

This chapter gives an overview over the design and the implementation of the Performance Bot, which utilizes PCM-to-LQN transformations to analyze the resulting LQNs fast. Section 3.1 describes the interface of the SQuAT Bots in general. Based on the interface of the SQuAT Bots, Section 3.2 derives the assumptions and requirements for the implementation of the Performance Bot. Section 3.3 then derives more specific tasks for the implementation of the optimization done by the Performance Bot from the specification of PerOpteryx. In addition, the architecture of the implementation is shown and explained. The same is done for the analysis performed by the Performance Bot, based on the Headless LQN Solver, in Section 3.4.

3.1. Interface

For the implementation of the SQuAT approach, which was presented in Section 1.3, a general interface has been defined. Figure 3.1 presents the basic idea of this approach, in which a SQuAT Bot has to work with an instance of an architecture. As the intention of the SQuAT approach is not to focus on a specific architectural model or quality attribute, there is no general implementation of a SQuAT Bot, but the interface *AbstractSQuATBot* instead. Nevertheless, the first phase of the project focuses on the development of a Modifiability Bot based on KAMP and a Performance Bot based on PerOpteryx. Both approaches assume PCM as the underlying architectural model. Because of that both bots implement the more specific *AbstractPCMBot*.

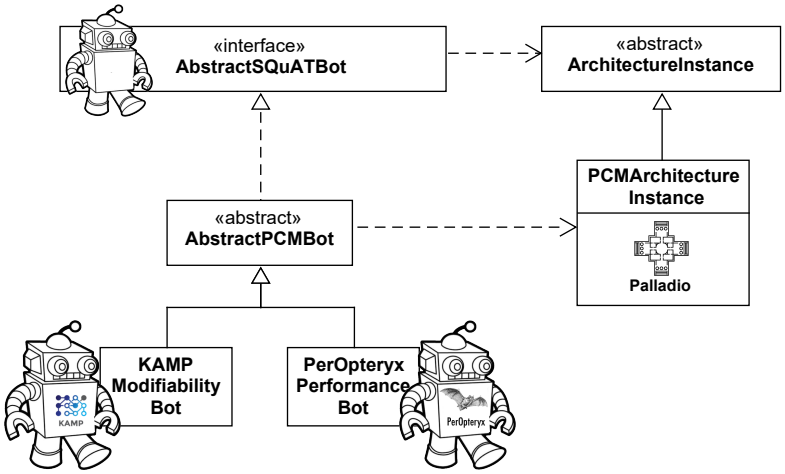

Figure 3.1.: SQuAT approach from the architecture view. [Pac+16]

Figure 3.2 shows more details about the design of the SQuAT Bot interface. To be able to execute the analysis phase and the searching phase, each bot has to implement the two methods *analyze* and *searchForAlt*. Both methods require the *ArchitectureInstance*, an initial candidate, to analyze or to optimize respectively.

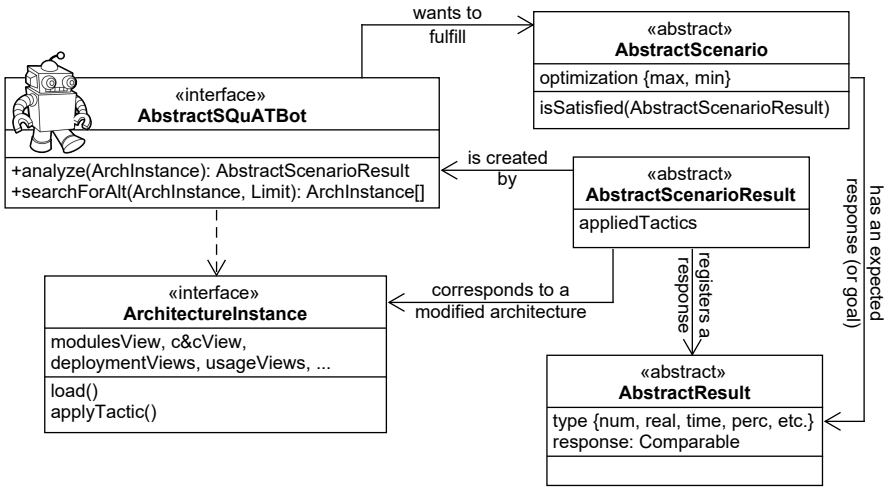

Figure 3.2.: Simplified architecture design of the SQuAT Bots. [Pac+16]

A response of the methods has to be registered, which is the computed value for a chosen quality attribute's metric. In addition, the optimization should also return the architectural models of the optimized candidates and the knowledge about the applied tactics necessary to generate the alternative candidates. As each bot wants to satisfy a

scenario, the *AbstractScenario* represents the goal for the bot. The scenario does not only determine the metric and the direction of the optimization, it must also contain an expected response, to decide whether the scenario is satisfied or not.

3.2. Assumptions and Requirements

First, a fundamental choice must be made on the approach of the Performance Bot. As outlined in Chapter 2, developing a new tool for architecture-based performance optimization would require much effort. Besides, it is not necessary, because several approaches already exist, as mentioned in Section 2.9. For the first evaluation of the SQuAT approach, a common input model would be beneficial, at least as long as no abstract model is available. PCM supports modifiability and performance and is designed to support model-to-model and model-to-code transformations. Thus, PCM is a promising starting point for the SQuAT project.

Further investigations based on the literature review conducted by Aleti et al [Ale+13] led to the conclusion that PerOpteryx seems to be the only existing approach which uses PCM instances as input. It would be necessary to apply additional model transformations to integrate other approaches. Besides, PerOpteryx is already validated and supports additional features, as described in Section 2.14. The performance tactics implemented in PerOpteryx are another reason to choose PerOpteryx as approach for the Performance Bot.

The integrated Palladio LQN Solver is chosen as analysis methods for two different reasons. As solvers are usually faster than simulators, choosing a solver will more likely prevent long waiting times for the other bots. The other reason is the Headless LQN Solver. This solver can be integrated into the Performance Bot as its analysis method. In addition, parts of it can be used as blueprint for the development of a Headless PerOpteryx.

It is possible to derive assumptions and requirements from the SQuAT approach described in Section 1.3 and the interface of its Bots described in Section 3.1. Assumptions are necessary to define the boundaries of the implementation and to show which functionalities are really necessary. PerOpteryx is a sophisticated approach for multi-objective architecture optimization, but only a part of its functionality has to be utilized in the Performance Bot. Thus, the assumptions restrict the implementation for the first version of a Headless PerOpteryx. They also restrict the necessary effort for the testing and evaluation of the Performance Bot.

The following three assumptions were identified for the Performance Bot:

A1 Only Performance: As each bot should contain the knowledge to analyze and optimize exactly one quality attribute, the Performance Bot must only be able to analyze and optimize software architectures with regard to performance metrics.

A2 Single Response: As each bot is assigned a single scenario and only a single response value is returned, the Performance Bot only optimizes a single objective at a time. This also means that the optimization is running with just one performance metric.

A3 Single Input: The Performance Bot optimizes only one initial candidate and not a whole starting population. In addition, it is not necessary to continue a stopped optimization.

Defining requirements is important to make a comparison between the intended behavior and the already available functionalities of the existing approaches. Tasks for the implementation can be derived from requirements, which are not fulfilled.

As the analysis and the optimization itself are intended be done by the existing approaches Palladio LQN Solver and PerOpteryx, essential parts of the functionalities are already available and must not be implemented again. However, it is even more important to assure the input and output of these tools is compatible to the interface of the Performance Bot. It is also necessary to consider the fact, that these tools are usually executed within the Eclipse environment. Finally, the tools must be configured and called by the Performance Bot and the results must be returned in the format of the defined result objects.

The applied tactics are consciously missing in the requirements for the optimization, because evolutionary algorithms make a lot of changes to a candidate throughout the whole optimization process. The tracing of all these changes is not supported by PerOpteryx and it would also be a lot of work to apply them again. Besides, for the first phase of the SQuAT project it is acceptable to just have the resulting PCM instances. As tracing of changes is not absolutely necessary for this phase of the project, it is not part of the requirements.

The following requirements for the optimization can be identified, based on the already mentioned categories:

$R1_{opt}$: Input Data

(1) The optimization should take a single PCM, which gets optimized.

(2) The optimization should take a single objective as input, to enable the extraction of the correct response values.

(3) The optimization should take path information, so the tool knows where the exported files should be located. In addition, the paths to files usually provided by the Eclipse environment have to be entered.

(4) As few as possible additional parameters have to be specified, default values should be used to keep the Performance Bot simple.

$R2_{opt}$: Environment

(1) A new environment has to be created to imitate the environment of Eclipse.

(2) The new environment takes path information from the input of the analysis.

$R3_{opt}$: Output Data

(1) The optimization must return a single response value for each returned candidate, depending on the chosen objective.

(2) The optimization should be able to return the PCM instances for the found candidate.

(3) The optimization should be able to return a specified number of best candidates, e.g., the best 10 candidates. As a bot needs alternatives for the negotiation phase, returning only the best candidate is not enough. On the other hand, returning all candidates would be too much.

$R4_{opt}$: Bot Implementation

(1) The results have to be converted into a proper result object.

(2) The optimization must be configured and PerOpteryx must be called by the Performance Bot.

Requirements for the analysis can be derived in a similar way than for the optimization, based on the same four categories:

$R1_{an}$: Input Data

(1) The analysis should take a single PCM, which gets analyzed.

(2) The analysis should take a single objective as input, to enable the extraction of the correct response value.

(3) The analysis should take path information, so the tool knows where exported files should be located. In addition, the paths to files usually provided by the Eclipse environment have to be entered.

(4) As few as possible additional parameters have to be specified, default values should be used to keep the Performance Bot simple.

$R2_{an}$: Environment

(1) A new environment has to be created to imitate the environment of Eclipse.

(2) The new environment takes path information from the input of the tool.

$R3_{an}$: Output Data

(1) The analysis must return a single response value, depending on the chosen objective.

$R4_{an}$: Bot Implementation

(1) The results have to be converted into a proper result object.

(2) The analysis must be configured and the LQN Solver must be called by the Performance Bot.

3.3. Implementation: Search for Alternatives

The state-of-the-art approach PerOpteryx has been chosen as the underlying approach for the implementation of the optimization. PerOpteryx has the advantage of using PCM, like the Modifiability Bot. The available LQN Solver is also considered as fast in comparison to other analysis methods, e.g., simulation. The optimization is therefore expected to run within an acceptable amount of time. Section 3.3.1 investigates in how far PerOpteryx is able to fulfill the requirements for the search for alternatives of the Performance Bot and derives the necessary tasks, to complete to integration of the approach into the Performance Bot. Section 3.3.2 then describes how the architecture of the Performance Bot and Headless PerOpteryx has been designed to fulfill these tasks.

3.3.1. Tasks

PerOpteryx is basically able to optimize a given PCM and is therefore suitable to undertake the task of searching for alternative architectures for the Performance Bot. The three performance metrics response time, throughput, and maximum CPU utilization can be used for the optimization. A comparison between the functionalities of PerOpteryx and the formulated requirements from Section 3.2 however reveal some differences. Figure 3.3 also illustrates this comparison.

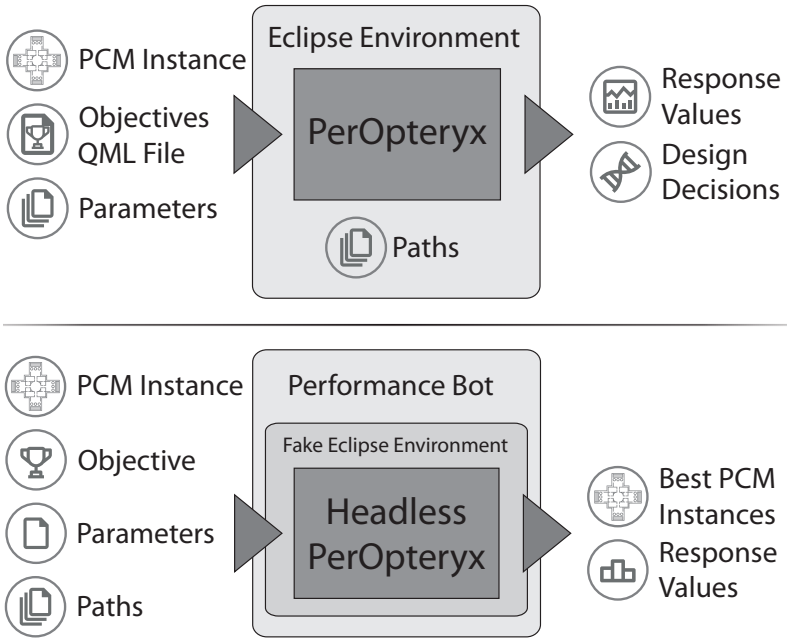

Figure 3.3.: Comparison between PerOpteryx (top) and the requirements for the optimization of the Performance Bot (bottom).

Requirement $\mathbf{R1_{opt}}(1)$ is already fulfilled for PerOpteryx, because the approach is based on PCM and also takes a PCM instance as input. However, the multi-objective approach is not fully compatible with requirement $\mathbf{R1_{opt}}(2)$. Usually a QML file has to be provided to specify which objectives should be activated for the optimization. For the Performance Bot it is only necessary to choose one of the three performance metrics, which does not require the wide range of options a QML file offers. As well as this requirement, requirement $\mathbf{R1_{opt}}(3)$ is not fulfilled, too. Most paths are provided by Eclipse, the analysis methods are also provided by a registry. The integration into Eclipse is also the main reason why PerOpteryx does not completely fulfill requirement $\mathbf{R1_{opt}}(4)$. All the parameters can be selected in the run configuration of Eclipse and there are many available options. On the one hand default values are also available there, on the other hand they will not be set without Eclipse. As a result, these values have to be set again in

a Headless PerOpteryx. In addition, many of the available parameters are not necessary for the Performance Bot.

The requirements $\mathbf{R2_{opt}}(1)$ and $\mathbf{R2_{opt}}(2)$ are obviously not fulfilled due to the integration of PerOpteryx into Eclipse. It is also not possible to simply call classes of PerOpteryx from a lower layer, because there are calls to Eclipse from multiple locations in the code. It is therefore necessary to make small changes to the code of PerOpteryx itself to provide the necessary path information in another way. However, the goal must be to keep these changes as small as possible, to reduce the manual effort to update Headless PerOpteryx to newer versions of PerOpteryx.

Another issue with PerOpteryx is the way it returns its results. PerOpteryx exports CSV files for all candidates, the populations and the Pareto-optimal candidates. These CSV files contain the desired response values and the choices made for the degrees of freedom. However, this only partially fulfills requirement $\mathbf{R3_{opt}}(1)$, because these values are needed as Java objects for the Performance Bot. In addition, requirement $\mathbf{R3_{opt}}(2)$ is completely unsatisfied, as only design decisions and no PCM instances are exported by PerOpteryx. For requirement $\mathbf{R3_{opt}}(3)$ it is also necessary to sort the results to extract a specified number of the best results.

As the requirements $\mathbf{R4_{opt}}(1)$ and $\mathbf{R4_{opt}}(2)$ can not be satisfied without an implementation for the Performance Bot, the following general tasks can be derived to fulfill the unsatisfied requirements:

$\mathbf{T1_{opt}}$: Develop a component to set the parameters for Headless PerOpteryx, which is simple and uses default values.

$\mathbf{T2_{opt}}$: Develop a component to convert the given objective to a simple QML file.

$\mathbf{T3_{opt}}$: Provide an environment, which imitates the Eclipse environment.

$\mathbf{T4_{opt}}$: Develop an export component, which extracts response values and PCM instances of a specific number of best candidates.

$\mathbf{T5_{opt}}$: Integrate Headless PerOpteryx into the Performance Bot method *searchForAlt*.

3.3.2. Architecture

Headless PerOpteryx is seperated from the Performance Bot to keep it extendable, so this part of the approach can be reused in other bots in the future. The tasks $\mathbf{T2_{opt}}$ and $\mathbf{T5_{opt}}$ are solved on the level of the Performance Bot, which is presented in Figure 3.4. The *PerformancePCMScenario* implements the *PCMScenario* and extends it with a *PerformanceMetric*, which is necessary to determine which response value should

be returned. To satisfy task $\mathbf{T2_{opt}}$ the *PerOpteryxQMLConverter* creates a simple QML file for the chosen *PerformanceMetric*. This file is used as input for Headless PerOpteryx, so the Performance Bot does not necessarily require a QML file as input.

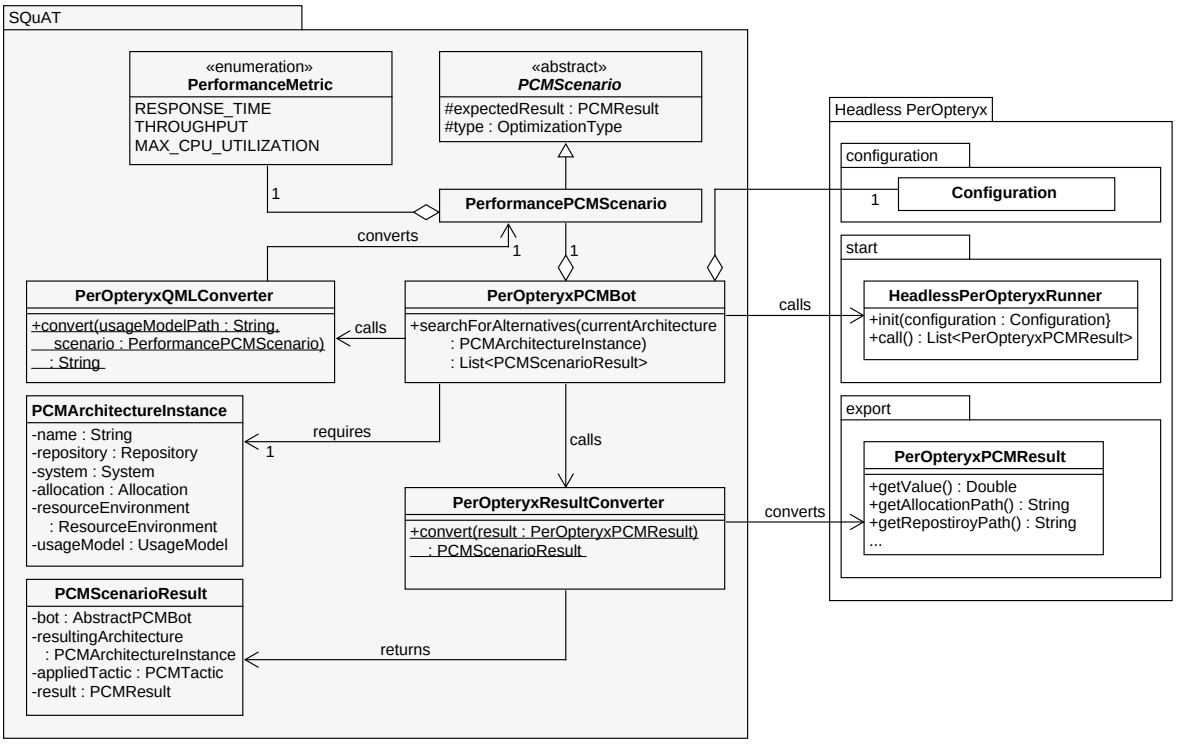

Figure 3.4.: Architecture of the Performance Bot for optimization (simplified).

Task $\mathbf{T5_{opt}}$ is satisfied through the integration of Headless PerOpteryx into the Performance Bot. A *Configuration* is necessary to construct the *PerOpteryxPCMBot* itself, because it already carries basic information about the paths and is then filled up by the Performance Bot. This configuration must be passed to the *HeadlessPerOpteryxRunner*, which is called to execute the optimization. To complete the task, the *PerOpteryxResultConverter* converts the results of Headless PerOpteryx to the desired format.

The *Configuration* is an important class in Headless PerOpteryx, because it provides all the values necessary to run the optimization. Figure 3.5 therefore presents the structure of the configuration package in more detail. The *Configuration* provides a more comfortable and structured interface to the raw data, which is converted into a configuration for PerOpteryx later. More specialized configuration classes assure a separation of concerns, e.g., *LQNSConfig* is only responsible for the configuration of the LQN Solver. They also provide default values as far as possible. These more specialized configurations can be accessed through the general *Optimization*. Finally, the PerOpteryx class *DSEWorkflowConfiguration* can be build from the configurations

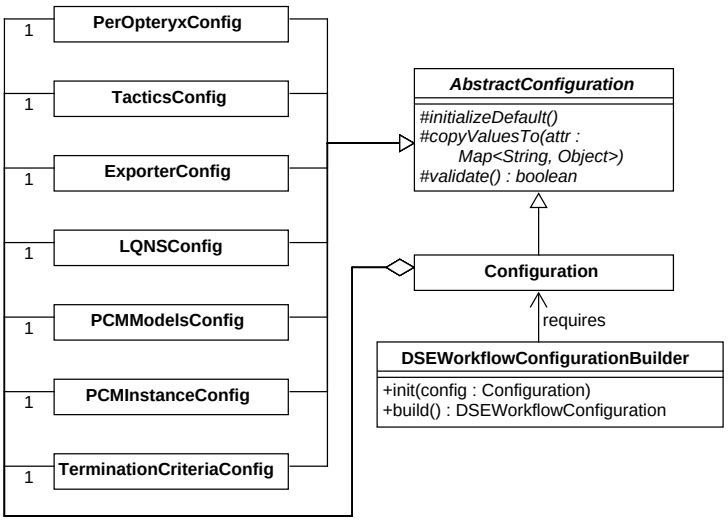

Figure 3.5.: The configuration package in Headless PerOpteryx (simplified).

and passed to the optimization itself. So in conclusion, this package assures task $\mathbf{T1_{opt}}$ is fulfilled by providing a simple and structured interface with default values for Headless PerOpteryx.

In Figure 3.6 Headless PerOpteryx is presented in more detail. The central interface of this tool is the *HeadlessPerOpteryxRunner*, which is also called by the Performance Bot. A part of the Eclipse environment is imitated by *PalladioEclipseEnvironment*, which is inspired by the implementation of the Headless LQN Solver. While the Headless LQN Solver only sets up an environment for Palladio, additional efforts are made for PerOpteryx in the *PerOpteryxEclipseEnvironment*. Both classes register paths and factories, which are usually provided by the Eclipse environment. The *PerOpteryxEclipseEnvironment* also takes care of the *ExtensionRegistry*, which provides the correct analysis method for the optimization.

Several classes from PerOpteryx had to be overwritten to be able to run it without the original Eclipse environment, like the *MyPerOpteryxJob*. This class runs several other jobs to make preparations and then kicks off the optimization process in *MyOptimizationJob*. The *Opt4JStarter* then runs the optimization with Opt4J. Many classes in the middle layer of PerOpteryx had to be overwritten, often in order to change or add only one or a few lines of code. No changes have been made to the optimization process of Opt4J, including the *Opt4JStarter*. Only the *SequentialExecutionInjector* is necessary to prevent Opt4J from running a multi-threaded optimization, which leads to wrong results and exceptions. The described architecture is able to run PerOpteryx without the original Eclipse environment and it therefore fulfills task $\mathbf{T3_{opt}}$.

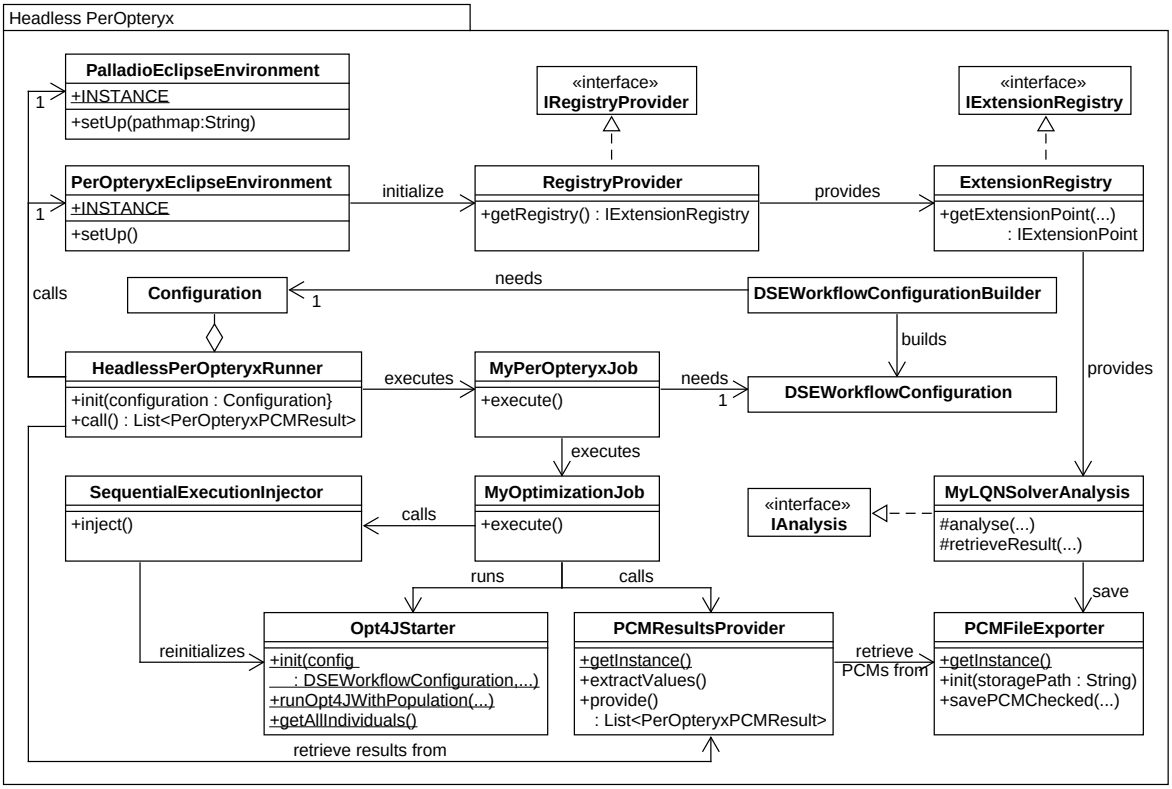

Figure 3.6.: Architecture of Headless PerOpteryx (simplified).

The remaining task $\mathbf{T4_{opt}}$ is fulfilled by the classes *PCMFileExporter* and *PCMResultsProvider*. The *PCMFileExporter* saves the PCM instances of already analyzed candidates to the hard disk. As soon as the optimization terminated, the *PCMResultsProvider* extracts the response values of all candidates. In addition, it sorts the results and provides the desired amount of best alternative candidates.

In conclusion, the presented design enables the Performance Bot to search for alternatives by utilizing the optimization pipeline of PerOpteryx. In combination with the implemented extensions, it also fulfills all the defined requirements under the formulated assumptions from Section 3.2.

3.4. Implementation: Analyze

One possibility would be to use the Headless PerOpteryx approach described in Section 3.3 for the analysis of a single candidate. If the optimization is run for only one iteration with one candidate, it is actually an analysis. However, this approach would

contain a large overhead due to the setup of the whole optimization pipeline, which is not needed in this case. The already existing Headless LQN Solver, mentioned in Section 2.13, can be considered as promising approach for this part of the Performance Bot, as it is already able to run without Eclipse. Section 3.4.1 describes which tasks are necessary to integrate this approach into the Performance Bot. Section 3.4.2 then describes which extensions have to be made in the architecture to fulfill these tasks.

3.4.1. Tasks

A comparison between the Palladio LQN Solver and the analysis method of the Performance Bot is illustrated in Figure 3.7 and reveals some differences. However, the Headless LQN Solver already solved some of these compatibility issues. Especially the requirements $\mathbf{R2}_{\mathbf{an}}(1)$ and $\mathbf{R2}_{\mathbf{an}}(2)$ are already fulfilled, because the Headless LQN Solver already sets up the environment.

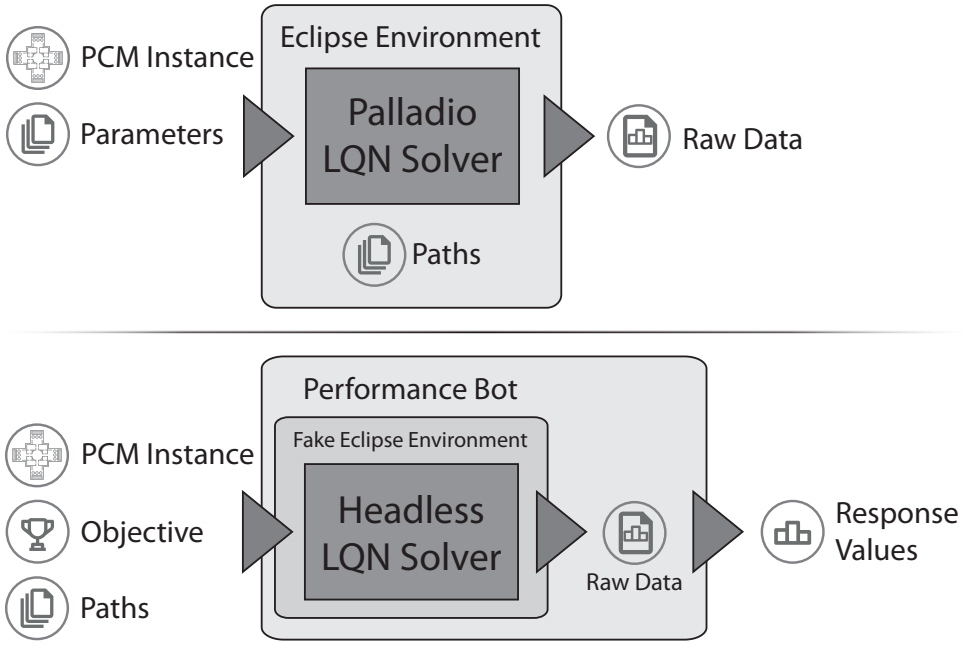

Figure 3.7.: Comparison between the Palladio LQN Solver (top) and the requirements for the optimization of the Performance Bot (bottom).

As the input for the Headless LQN Solver is still a PCM instance and the PCM-to-LQN transformation is applied internally, requirement $\mathbf{R1}_{\mathbf{an}}(1)$ is also fulfilled. Requirement $\mathbf{R1}_{\mathbf{an}}(2)$ is basically not fulfilled, but this is not a big issue at all. The LQN Solver does always perform a full analysis, so the objective is only required to choose the response to return in the end. Requirement $\mathbf{R1}_{\mathbf{an}}(3)$ and $\mathbf{R1}_{\mathbf{an}}(4)$ on the other hand are already

fulfilled through the design of the Headless LQN Solver. The paths can be given as input and the solver can be executed with predefined values.

The biggest problem is the output of the results by the Headless LQN Solver, because it only delivers a file, which contains a large number of computed values for the given PCM instance. The response values for the desired performance metrics still have to be computed from these values. Because of this, requirement $\mathbf{R3_{an}}(1)$ is obviously not fulfilled.

As the requirements $\mathbf{R4_{an}}(1)$ and $\mathbf{R4_{an}}(2)$ can not be satisfied without an implementation for the Performance Bot, the following general tasks can be derived to fulfill the unsatisfied requirements:

$\mathbf{T1_{an}}$: Develop an export component, which computes the response value for the given objective from the raw output of the solver.

$\mathbf{T2_{an}}$: Integrate the Headless LQN Solver into the Performance Bot method *analyze*.

3.4.2. Architecture

Figure 3.8 shows the architecture for the analysis of the Performance Bot. The Headless LQN Solver already provides the classes and the functionality for the configuration of the solver. After the required values have been passed, the analysis can simply be executed.

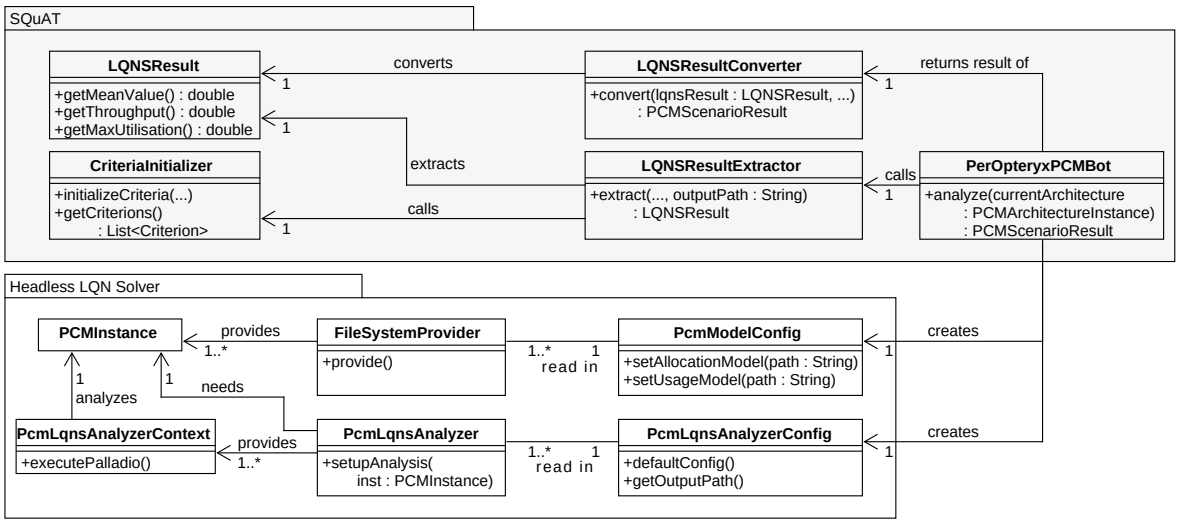

Figure 3.8.: Architecture of the Performance Bot for analysis (simplified).

A *LQNSResult* is then extracted from the raw output file by the *LQNSResultExtractor*. As PerOpteryx is already able to extract the required response values, it is possible to reuse code from PerOpteryx for this classes. The *CriteriaInitializer* is required to run the PerOpteryx code without making major changes to it. Finally, the *LQNSResultConverter* just has to choose the values for the given performance metric and converts the result to the correct format for the Performance Bot.

In conclusion, not much effort is required to solve the tasks $\mathbf{T1_{an}}$ and $\mathbf{T2_{an}}$. This makes the integration of the Headless LQN Solver into the Performance Bot simple, especially because the Headless LQN Solver already fulfills most of the formulated requirements from Section 3.2.

Chapter 4

Evaluation

This chapter describes the evaluation of the Performance Bot presented in Chapter 3 and shows the suitability of the chosen approach. Section 4.1 describes the requirements for example systems, which are needed to evaluate the approach, and presents three example systems in more detail. The hypothesis and the most important information about the setup for the experiment are presented in Section 4.2. Section 4.3 gives an overview over the results of the experiment, which are subsequently analyzed in Section 4.4.

4.1. Example Systems

The evaluation of the Performance Bot requires an appropriate example system. This system will be analyzed in the experiment. Section 4.1.1 presents the requirements, which should be satisfied by such an example system. The Media Store Example from Section 4.1.2 and the Simple Tactics Example from Section 4.1.3 were both considered for this evaluation and are therefore presented in the following. The Extended Simple Tactics Example presented in Section 4.1.4 is a combination of the two other systems and overcomes the drawbacks of these systems.

4.1.1. Example System Requirements

The example system should fulfill some requirements to make the evaluation more meaningful and to assure the example system is also useful for the SQuAT project. Three primary requirements and one secondary requirement have been identified and are described in the following:

Complexity: The most important requirement is the complexity of the example system. Software systems in the real world usually consist of several components. The example system should therefore also contain at least five components and some dependencies to reflect the complexity of real software systems.

Solvability: Another important requirement is the solvability of the example system. The Performance Bot uses a transformation from PCM to LQN and afterwards solves the resulting LQN model. This method will not work for every system and especially not for every PCM feature. As a consequence the chosen example system should be a system that can be solved by this method.

Support for modifiability: Modifiability can not be modelled in default PCM and it is therefore not expected that an example system contains annotations for modifiability. However, it would be convenient to have an example system that can also be analyzed by the Modifiability Bot and shows some tradeoffs between performance and modifiability. This requires an example system that contains at least some degrees of freedom for modifiability and can be annotated later.

Support for other quality attributes: This is a secondary requirement, but the support of more quality attributes could make the example system more useful for future work, e.g., analysis with a Performance Bot and a Reliability Bot.

As creating PCM instances from scratch is an ambitious and time-consuming task, already existing examples should be used for the evaluation. There are some example systems available [Exa]. However, they had to be configured for the use with PerOpteryx and could not all be investigated in full detail. In conclusion, all applied tests failed, because they were not compatible with the current version of PCM or returned errors. Nevertheless, the tests of two example systems worked fine enough to consider them for the evaluation and are therefore described in the following sections. The Media Store Example was already successfully used for modifiability in the SQuAT project and the Simple Tactics Example was already configured for the use with PerOpteryx and the LQN Solver.

4.1.2. Media Store Example

The Media Store Example models a Web Music Store [SK16], where users can download and upload songs. The basic architecture (see Figure 4.1) is a three-tier architecture, which consists of a web-front-end, the application and a database running on its own server. Users can make requests to the *WebGUI*, which is the web interface of the application and will forward the requests to the *MediaStore* component. The songs itself are retrieved from the *AudioDB* and get watermarked by the *DigitalWatermarking* to

assure copy protection. Finally, the requested files are send to the user. The system also allows the upload of files, but this only occurs in 20% of all cases, in contrast to the 80% for downloading.

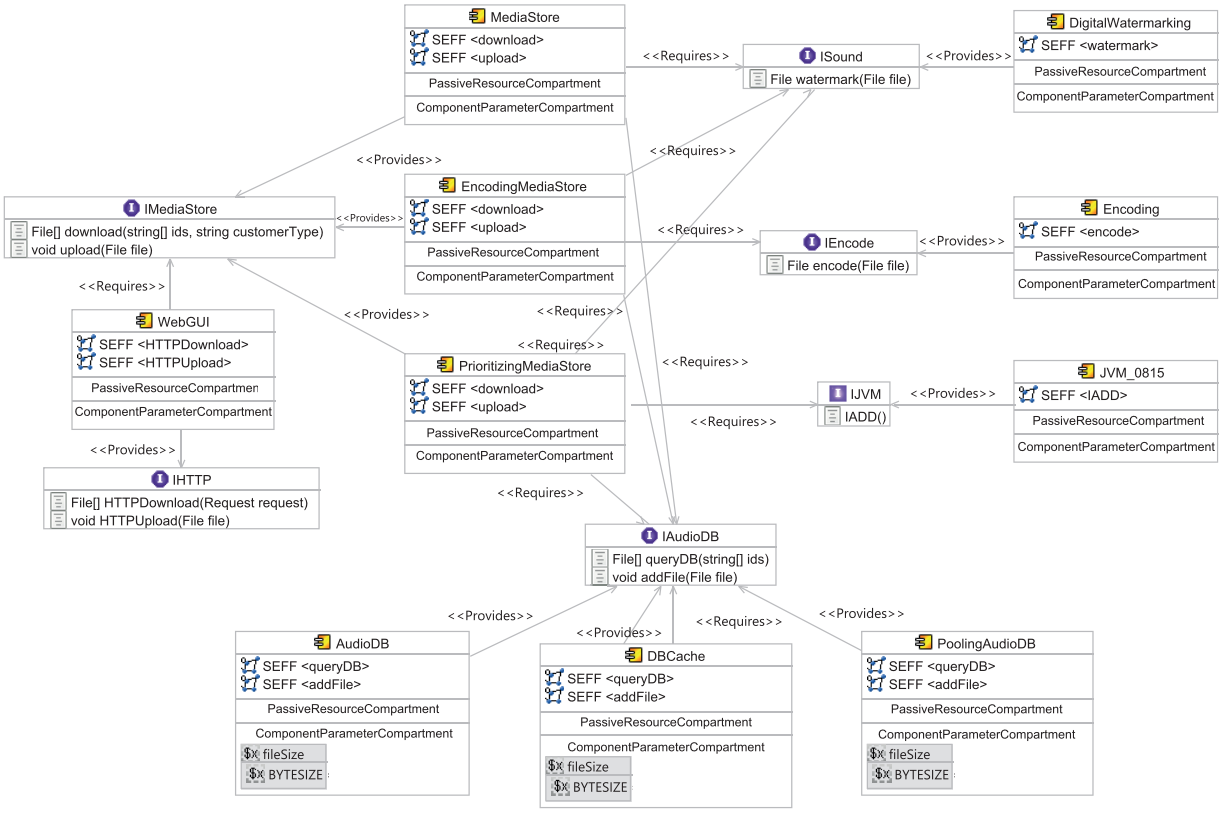

Figure 4.1.: Repository diagram of the Media Store Example.

This model also contains the three optional features Caching, Pooling, and Encoding. The *DBCache* keeps some songs in a cache, which reduces the number of required hard disk drive accesses, but requires additional computational effort for availability checks. Pooling requires the use of the *PoolingAudioDB* and can reduce the number of concurrent accesses to the hard disk drive, for the cost of having only a limited amount of concurrent transactions and thus possibly more waiting users. The *EncodingMediaStore* and the *Encoding* components can be used to reduce the size of high quality song files by reencoding them. The encoding causes additional computational efforts, but the reduced file size can lower the usage of the data link between the application server and the database server, as well as the hard disk usage of the database.

This example system contains 10 different components, but depending on the used features only 4 to 5 of them are used simultaneously. Nevertheless, this system can be considered as complex enough for the evaluation. The optional features of the

Media Store also lead to the assumption, that this model can be annotated to get some tradeoffs between performance and modifiability. Degrees of freedom mainly grounded on hardware changes are supposed to have only a small or even no impact on the modifiabiliity of the system, thus changes to the components itself must be made. The optional features Caching, Pooling, and Encoding include several degrees of freedom from the component selection category, which can likely have an impact on modifiability.

The most critical issues with this model occurred for the second requirement: the solvability. PerOpteryx did not return exceptions, but the solver seemed not to be able to solve this model. Changes in the configuration and the model itself did not resolve this issue. It is therefore assumed, that this model contains PCM features the solver can not solve or another restriction of the solver exists, e.g., the computed utilization of a CPU exceeds 100%. The possibility that this model was not configured in the right way can also not be completely denied. The exact reason might be unknown, however this example can not be solved and therefore this example system can not be used for the evaluation.

4.1.3. Simple Tactics Example

The Simple Tactics Example was originally part of the Palladio Example collection and models a business trip management system (see Figure 4.2). Despite from the booking and managing different payment options, this also includes making reimbursements. For the Simple Tactics Example an open workload is assumed. As already implied by the name, this example system is simple and contains only four components. The user can call the method *plan* of the *BusinessTripMgmt* component. The parameters determine if the user wants the system to make a reimbursement or a booking. The *BookingSystem* is then called to start a booking. In any case the *PaymentSystem* is called in this process to make payments or reimbursements.

This system provides some degrees of freedom, mainly related to the hardware. The components can be allocated to three different servers and the CPU clock rate can be changed. In addition, the *QuickBooking* component is available as alternative for the *BookingSystem*.

One benefit of the Simple Tactics Example is the solvability of this example system. This model was already configured for the use with PerOpteryx and the LQN solver, because of that there were no bigger issues detected in the tests. Another benefit of this example system is that it can also be used for optimization of reliability and costs, because the model is already annotated for this tasks. Thus, this example system fulfills at least one primary requirement and the secondary requirement for the evaluation.

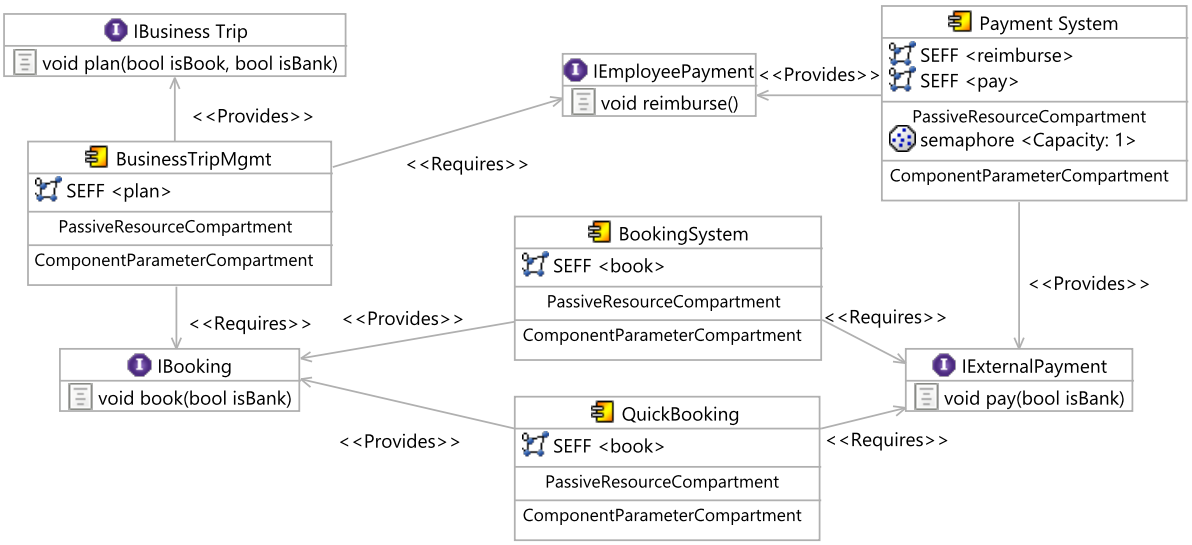

Figure 4.2.: Repository diagram of the Simple Tactics Example.

The support of modifiability is nevertheless expected to be low, because there are mainly degrees of freedoms with influence on the hardware of the system. One alternative for a component is unlikely to be sufficient for providing many tradeoffs between performance and modifiability. However, the most critical issue with the Simple Tactics Example is the complexity. Only three components are used simultaneously and there is only one initial method to call. In conclusion, this example system does not fulfill all requirements and can therefore not be used in the evaluation.

4.1.4. Extended Simple Tactics Example

The Extended Simple Tactics Example is a combination of the Simple Tactics Example and parts of the Media Store Example. On the one hand the Simple Tactics Example has the advantage of being solvable and offers support for reliability and cost, while on the other hand the Media Store example is more complex and likely to provide support for modifiability in the future. Thus, the idea was to extend the Simple Tactics Example with new components, which are based on the Media Store Example to create an example system, which fulfills all the requirements at the same time.

Different components often differ in their modifiability values. Thus, the Modified Simple Tactics Example contains some alternative components, which can make tradeoffs necessary. The extended version also saves information about booked business trips to a database, similar to the one from the Media Store Example. If a business trip is booked, a new file has to be added to the database. If a reimbursement is made,

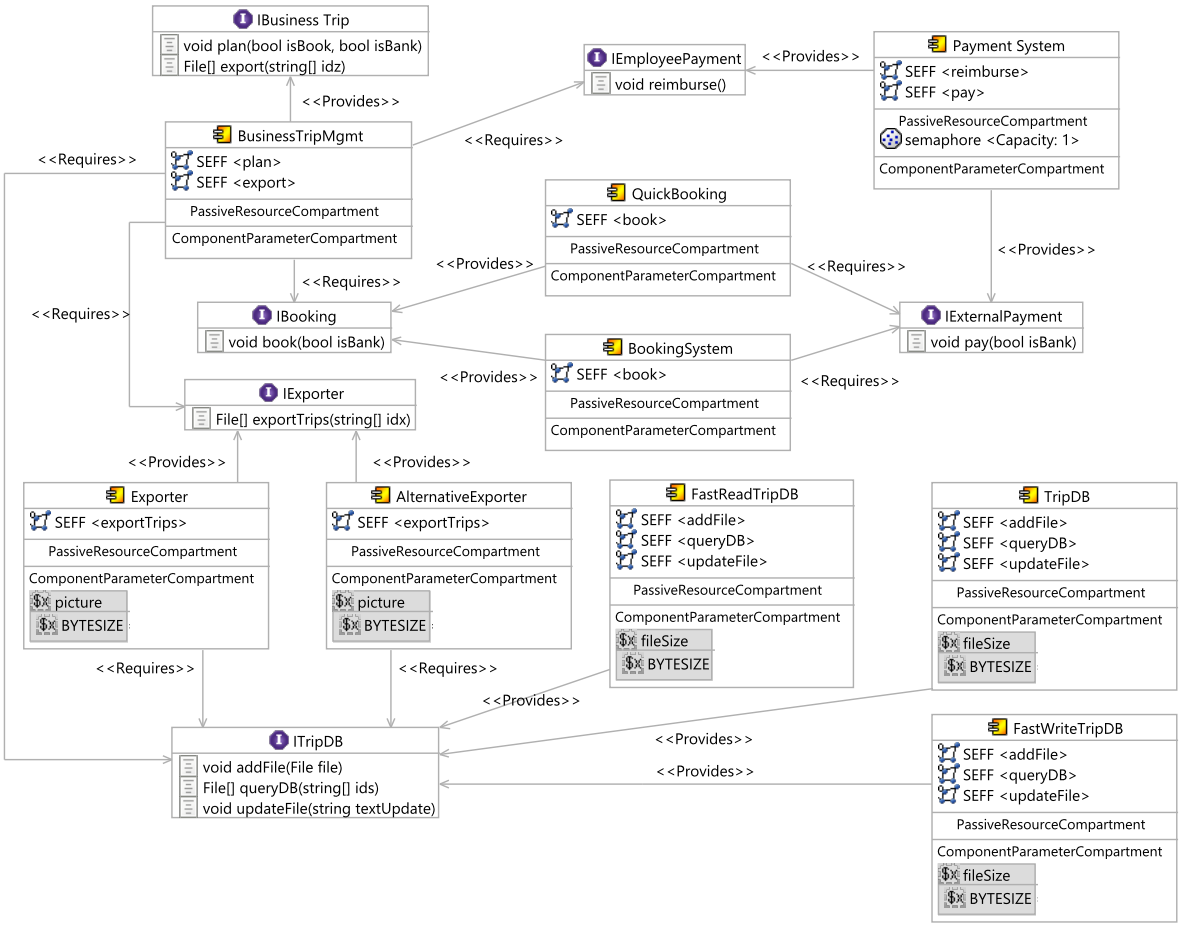

Figure 4.3.: Repository diagram of the Extended Simple Tactics Example.

the file in the database has to be updated. The default database component is the *TripDB* component. This component can be replaced by the *FastReadTripDB* or the *FastWriteTripDB*. The *FastReadTripDB* improves the speed of reading from the database for the cost of lower writing speed. The *FastWritingTripDB* is the counterpart to this component. The database offers the users the possibility to export information about booked business trips and the payment as a PDF file. The information is formatted by the *Exporter* in human readable form, because the database only stores raw data. In addition, logos have to be added to the PDF. The *AlternativeExporter* stores the logos in a compressed version to improve the speed of loading the file. Figure 4.3 also shows an overview over the available components and interfaces of the extended example system.

The Simple Tactics Example system already contained three servers for the allocation of the components. With the addition of the database, a new database server has been added to them. The database server contains a much faster HDD than the other servers

and uses the same network connection as the other servers. As a result, the Modified Simple Tactics Example provides many degrees of freedom for the optimization of the system, which are identified automatically by PerOpteryx. The components can be allocated to all servers. In addition, the speed of the CPU and HDD of all four servers can be changed. However, the most important degrees of freedoms are from the category of component selection. There is one alternative for the *BookingSystem*, one alternative for the *Exporter* and two alternatives for the *TripDB*.

The size of the example system is considered big enough, because five components are always used in every configuration. In addition to the method *plan* from the Simple Tactics Example, 10% of the users also use the *export* method of the system, which further increases the complexity of the system.

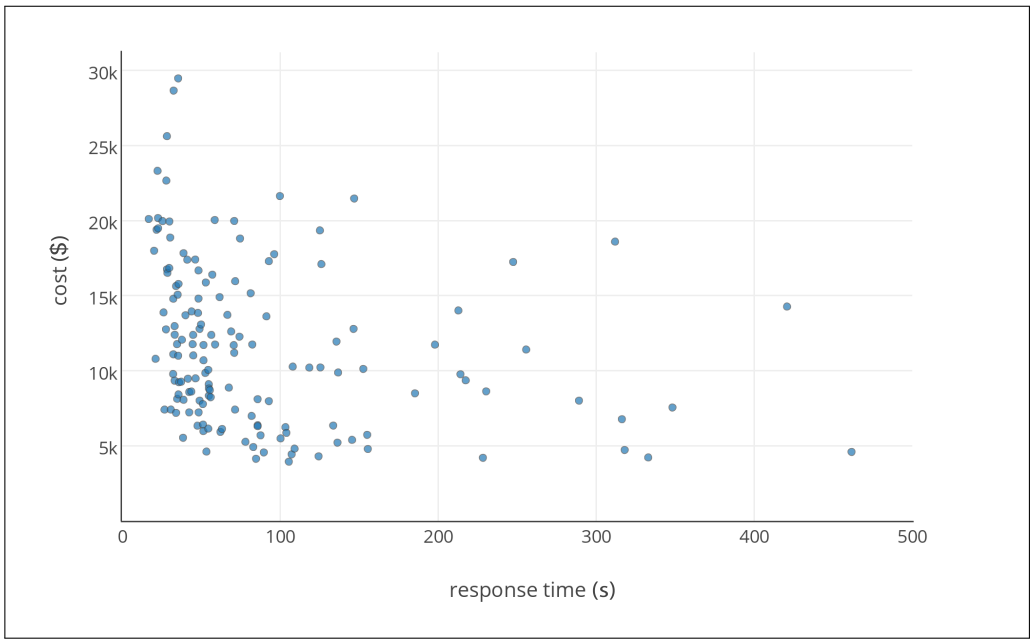

Figure 4.4.: All found candidates with response time lower than $500\,\mathrm{s}$ for the optimization of the Extended Simple Tactics Example.

A first test was run with PerOpteryx on the Extended Simple Tactics Example to evaluate the solvability. PerOpteryx was configured to make a run with 20 iterations and 200 individuals per generation. No tactics were used and the crossover rate has been set to 50%. To demonstrate the possibility to support tradeoffs, response time and cost were optimized. PerOpteryx returned a total number of 2100 candidates, including the initial candidate. Only 153 generated candidates and the initial candidate could be solved, which is only 7.3% of all found candidates. Figure 4.4 presents all the successfully evaluated candidates with a response time lower than 500 and Figure 4.5 shows those which are Pareto-optimal. The low number of solvable candidates seems to

be a problem on first glance, but further tests showed, that the most frequent reason for this behavior was the allocation of the components. It also has to be considered, that the LQN solver fails, if the utilization of a CPU exceeds 100%. Such a bottleneck would in general not be beneficial for the software system, thus it can be concluded, that most of these candidates would probably not be Pareto-optimal and can be ignored. It is at least possible to use the Extended Simple Tactics Example, even though most of the candidates have no response value.

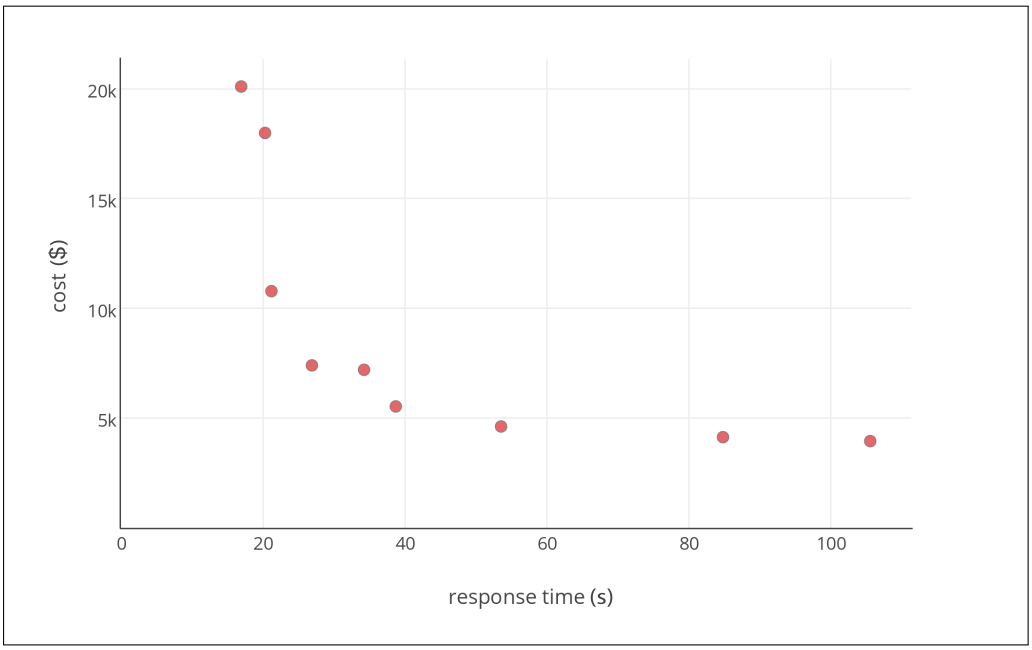

Figure 4.5.: Pareto-optimal candidates for the optimization of the Extended Simple Tactics Example.

Figure 4.4 and Figure 4.5 also show that it is at least possible to find tradeoffs between performance and cost. The technique of cost evaluation is partially similar to the techniques for modifiability evaluation, because both assign values to components. Future extensions of this example system to support modifiability therefore are assumed to provide tradeoffs between performance and modifiability. This example system could even be more suitable for this task than the Media Store Example, which only contains three optional features. In addition, cost and reliability can be optimized, too.

In conclusion the Extended Simple Tactics Example fulfills all requirements for the use as example system in the evaluation part of this work. Additionally it has the potential to be used in a later case study for the SQuAT project.

4.2. Experimental Setup

Every experiment needs preparations. For the evaluation of the Performance Bot all experiments are described in this section. The first step of every experiment is to find a hypothesis and ways to determine whether it is confirmed or rejected. This step is described in Section 4.2.1. Section 4.2.2 then describes the experiment to evaluate the analysis part of the Performance Bot. Two more experiments for the optimization part of the Performance Bot are described in Section 4.2.3 and 4.2.4. Finally, Section 4.2.5 summarizes the most important information about the general settings of the approaches and the environment in which the experiments are conducted.

4.2.1. Hypothesis

For the evaluation the goal is to show, that the developed Performance Bot can compete with other state-of-the-art tools for performance architecture optimization. As PerOpteryx has already been evaluated and compared to other tools, it is sufficient to show, that the Performance Bot does not perform worse than PerOpteryx. On the other hand, the Performance Bot is not expected to perform better, because the underlying approach and technology is the same.

The following concepts have been identified to be useful for the evaluation:

Quality: It is not possible to show the correctness of an approach by testing and without a mathematical proof. However, it is highly unlikely to get two identical values for a complex model by pure coincidence. For the analysis such a comparison is obviously possible. For the optimization at least the computed values for the initial candidate can be compared, because this one is not randomly generated. For optimization in general, which is a single-objective optimization for performance, the results for the best candidates after several runs can be statistically analyzed and compared based on their distributions.

Convergence: Not only the quality of the results is important, it is also important how fast a sufficient quality can be reached. It is assumed that the optimal candidate will not improve much, as soon as it is close to the real Pareto-front. It is therefore possible to measure the number of iterations at which the current optimal candidate is not improved much anymore. This number of iterations can be used to compare different approaches.

Runtime: Comparisons of runtimes for several runs for the same optimization job are another way to compare two approaches. The runtimes for the optimization in PerOpteryx and the Performance bot are basically assumed to be similar. However,

both approaches are not completely the same, e.g., PerOpteryx does not export PCM instances. Considering the uncertainty of runtimes, the differences are nevertheless not expected to be big.

The Performance Bot consists of two parts, the analysis and the optimization. While the analysis is the smaller and simpler part to evaluate, the evaluation of the optimization is more difficult. The results generated by evolutionary algorithms are random, therefore they are unlikely to be the same in two independent runs, even for the same model. As a result, the results of both parts of the Performance Bot have to be treated differently in the evaluation. Three different experiments are derived from the presented concepts for the Performance Bot and are described in the following sections.

4.2.2. Quality of Architecture Analysis

The first experiment evaluates the quality of the analysis approach. For this experiment the results for the performance metrics response time, throughput, and maximum CPU utilization of three different approaches are compared. The first investigated approach is the analysis method of the Performance Bot, which utilizes the Headless LQN Solver. In addition, the results of PerOpteryx and the optimization method of the Performance Bot based on Headless PerOpteryx are investigated with only the initial candidate and no further generation of candidates. The investigation of the Performance Bot's optimization method is made to ensure the analysis inside of it works correctly, too. PerOpteryx is investigated, because it uses the Palladio LQN Solver and is expected to deliver the correct results for a comparison. The Palladio LQN Solver can't be used for this purpose, since it does only export some raw values and does not compute the values for the required performance metrics.

The Extended Simple Tactics Example is used for this experiment with the default components selected. All three performance metrics are measured in one single run for PerOpteryx. The Performance Bot is only able to report one result, thus it is executed three times for both methods. The runtime is also reported to get a basic idea of how long one run takes. However, a detailed investigation of the runtime for the analysis of only one PCM instance is not planned, because it is expected to be short and several factors can have a huge influence on the results. The runtime of the Palladio LQN Solver will also be reported.

The values are expected to be the same for all three approaches and for all three considered metrics. It is already logical to prefer the Headless LQN Solver over Headless PerOpteryx with one iteration and one candidate to analyze a PCM instance, because the Headless LQN Solver does not have to initialize the optimization for this purpose.

Nevertheless, it would be good to have a runtime measurement to estimate the benefit of this choice.

4.2.3. Quality and Runtime of Architecture Optimization

The second experiment evaluates the quality and the runtime of the optimization part of the Performance Bot. The results of the optimization method of the Performance Bot and PerOpteryx are compared. According to the scenarios in SQuAT, it is expected that one instance of the Performance Bot will be used to optimize one single performance metric. For this experiment response time is chosen as the investigated performance metric. The goal for this part of the experiment is to show that the Performance Bot optimizes candidates. Additionally, the optimization is expected to deliver best candidates with more than 99% of the improvement PerOpteryx is able to reach.

In addition to the response value, the runtime is measured and investigated. In contrast to the analysis, one run is expected to take much longer and therefore the influence of other uncontrollable factors on the runtime is expected to decrease. As the Performance Bot has to export the PCM instances and sort the results in order to return a specific amount of best candidates, it is expected to be slower than PerOpteryx itself. Thus, a decrease of 5% in runtime in comparison to PerOpteryx is considered to be still satisfying.

The Extended Simple Tactics Example gets optimized 30 times for both approaches in this experiment. In each run a total of 20 iterations is performed and the size of the population is limited to 200. The response value of the best candidate in every run is stored and a distribution is computed for both tools.

4.2.4. Convergence of Architecture Optimization

A stop criteria implemented in PerOpteryx and Headless PerOpteryx can be used for the third experiment conducted in the evaluation. If the response value of the best candidate is considered insignificant, the optimization will stop before it reaches the maximum of the allowed iterations. Insignificance in this stop criteria is described by an amount of iterations in which the best candidate is only improved by less than a specific percentage. For this experiment the amount of iterations is set to 6 and the upper limit for the improvement to 1%.

As in the experiment before, the Extended Simple Tactics Example is optimized 30 times for response time. This time a maximum of 50 iterations is chosen and the population is limited to 300. The amount of executed iterations is measured and decreased by 6, due

to the numbers of additional runs the tools had to execute to determine no significant change was made anymore. Again, the goal is to be at least 99% as fast as PerOpteryx in finding nearly fully optimized candidates.

4.2.5. Experimental Settings and Environment

For all executions of optimization tasks a crossover rate of 50% and tactics probability of 60% is used. All tactics are activated with default values, except for the antipattern detection, because this one is not able to change anything on the example system and reported problems on the console.

The runtime measurements for PerOpteryx and the Palladio LQN Solver are taken from the console output of these tools and are called “workflow execution time”. It has to be mentioned, that this time is likely to be not the complete runtime and some jobs, e.g., the export, might be not part of this measurement. However, it is difficult to measure them in a better way, because booth tools are integrated into the Eclipse IDE. The measurements for the Performance Bot are all made for the full call of the method. This includes the calls to the headless tool libraries and result conversions inside of the Performance Bot’s project.

The first runtime measurement of a tool gets always discarded, because their values often turn out to be much higher than the following results. The reason for this could be caching or other improvements made by the operating system. As the measurements can not be protected from influences through these effects, allowing them for all measurements is the more practical solution.

The evaluation is performed on a desktop computer with a quad core CPU, $3.4\,\mathrm{GHz}$ each core, and $16\,\mathrm{GB}$ RAM, $2\,\mathrm{GB}$ of them are allocated to the virtual machine of Java. After every measurement the garbage collector is run from the Eclipse IDE. During performance measurements the computer is not used for other purposes.

4.3. Experimental Results

The results for the measurements and the computed descriptive statistics data are summarized in this section. Section 4.3.1 presents the results for the experiment on architecture analysis. Section 4.3.2 then describes the results for the experiment on the quality and runtime of architecture optimization. The results of the third experiment on the convergence of the approaches is presented in Section 4.3.1.

4.3.1. Quality of Architecture Analysis

Table 4.1 presents the analysis results computed by the investigated approaches for the initial candidate of the Extended Simple Tactics Example. The response time for this candidate has been determined to be $105.5573\,\mathrm{s}$. The throughput of the candidate is $0.5\,\frac{\mathrm{tasks}}{\mathrm{s}}$ and the maximum CPU utilization is 0.9222 It has to be mentioned that all tools reported the same values.

	PerOpteryx	Performance Bot		Palladio
		Optimization	Analysis	LQN Solver
Response Time	$105.5573\,\mathrm{s}$	$105.5573\,\mathrm{s}$	$105.5573\,\mathrm{s}$	-
Throughput	$0.5\,\frac{\mathrm{tasks}}{\mathrm{s}}$	$0.5\,\frac{\mathrm{tasks}}{\mathrm{s}}$	$0.5\,\frac{\mathrm{tasks}}{\mathrm{s}}$	-
Max. CPU Util.	0.9222	0.9222	0.9222	-
Runtime	$1.322 \pm 0.090\,\mathrm{s}$	$2.754 \pm 0.049\,\mathrm{s}$	$1.531 \pm 0.064\,\mathrm{s}$	$0.392 \pm 0.037\,\mathrm{s}$

Table 4.1.: Results for different tools analyzing the initial candidate of the Extended Simple Tactics Example.

The table also reports the measured runtime for 30 runs. As expected, the runtime for the Performance Bot methods was nearly the same for all performance metrics. Because of this, only the runtime for the response time run is shown in the table. In addition to the shown values, Palladio and PerOpteryx also reported a value for the analysis only. For the Palladio LQN Solver the analysis took $0.254 \pm 0.023\,\mathrm{s}$ of the $0.392 \pm 0.037\,\mathrm{s}$ and for PerOpteryx it took $1.129 \pm 0.085\,\mathrm{s}$ of $1.322 \pm 0.090\,\mathrm{s}$. The additional time comes from other tasks, e.g., loading and validating the model.

4.3.2. Quality and Runtime of Architecture Optimization

A statistical analysis of the results for the second experiments results in a mean of $18.158\,\mathrm{s}$ for the response time of the best found architecture after the optimization for PerOpteryx. The mean response time for the Performance Bot is $17.366\,\mathrm{s}$. While the mean value is slightly better for the Performance Bot, also the standard deviation $1.524\,\mathrm{s}$ is lower than the standard deviation $1.908\,\mathrm{s}$ for PerOpteryx. The results are also presented in Figure 4.6. The best candidate was found by PerOpteryx and it has a response time of $14.1590\,\mathrm{s}$. This is a total improvement of $91.3983\,\mathrm{s}$ on the response time compared to the initial candidate. In Section 4.2.3 the decision was made to have the goal to reach 99% of PerOpteryx's improvement with the Performance Bot, which would be at least $86.5253\,\mathrm{s}$ for the mean value.

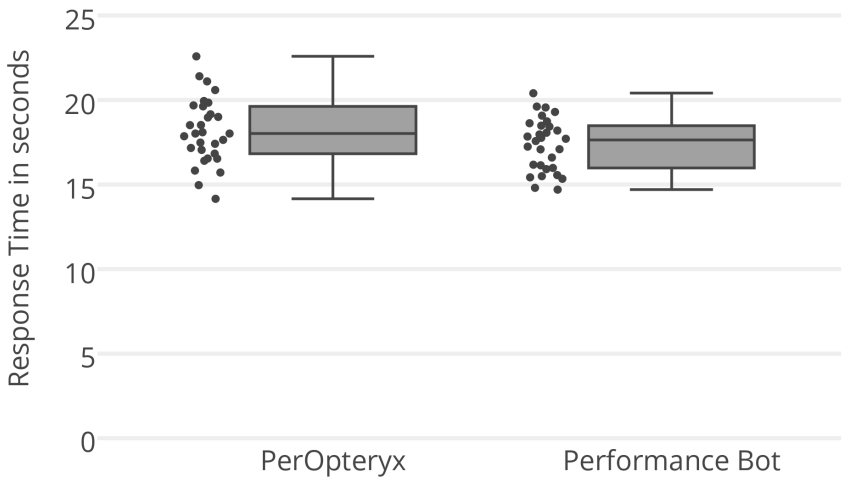

Figure 4.6.: Boxplots for the response times of the best found candidates.

Figure 4.7 presents the results for the runtime of the optimization. A mean of $229.421\,\mathrm{s}$ was determined for PerOpteryx and a mean of $226.596\,\mathrm{s}$ for the Performance Bot. The standard deviation for PerOpteryx is $12.464\,\mathrm{s}$ and for the Performance Bot $6.089\,\mathrm{s}$. It has to be added, that the values for the PerOpteryx runs seemed to increase with the number of runs, but the reason for this pattern in the measurement is unknown. Based on the goal defined in Section 4.2.3 for the Performance Bot to run only 5% slower than PerOpteryx, this would allow the Performance Bot to be more than $10\,\mathrm{s}$ slower in any case.

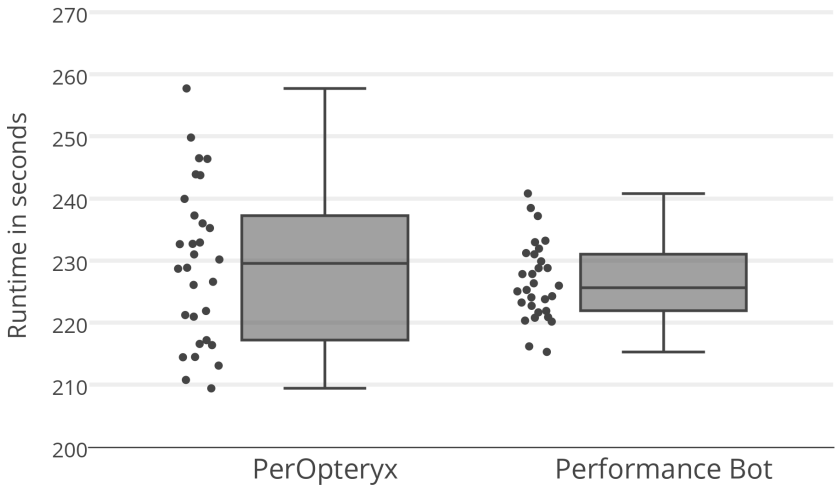

Figure 4.7.: Boxplots for the runtime of the optimization.

4.3.3. Convergence of Architecture Optimization

The results for the convergence are presented in Figure 4.8. Most of the runs took less than 10 runs to find a candidate near to the optimal solution. There are even some runs, which needed only one or two runs to fulfill this task. The mean for PerOpteryx is 4.033 and for the Performance Bot 6.067. However, the standard deviations can be considered high compared to the mean with 3.573 for PerOpteryx and 4.234 for the Performance Bot.

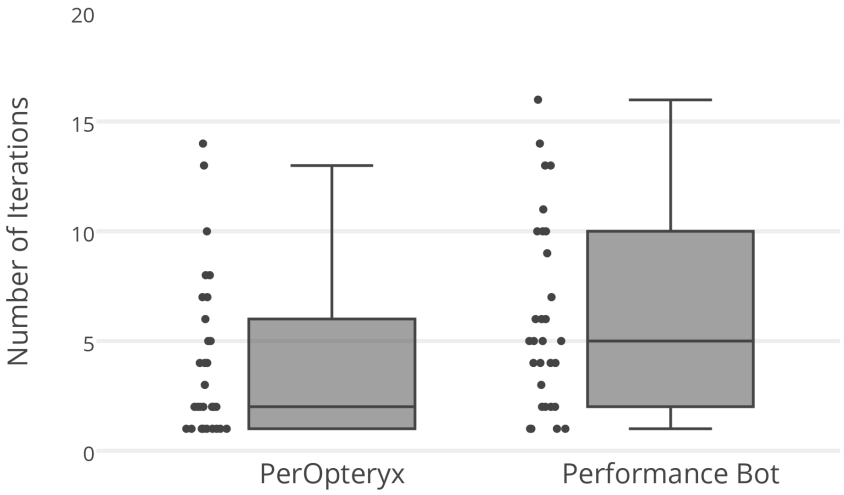

Figure 4.8.: Boxplots for the number of iterations at which the last significant change on the response time of the best found candidate occurred.

4.4. Result Interpretation

The results for the analysis method of the Performance Bot show that the Headless LQN Solver and Headless PerOpteryx report the same values as PerOpteryx. This demonstrates that the chosen approach is likely to be configured correctly and able to analyze PCM instances correctly. In addition, the results for the runtime support the assumption, that it was a good decision to choose the Headless LQN Solver for the analysis. Nevertheless, the runtimes are short and difficult to compare due to the different measurement methods. Thus, a more detailed analysis is not possible, at least with this the used measurement methods. Anyhow, it is possible to get an idea of the runtime for a single analysis, which lies approximately between $0.2\,\mathrm{s}$ and $2.0\,\mathrm{s}$ for the used model.

On the results for the experiment about the quality and runtime of the architecture optimization a non-inferiority test based on confidence intervals was applied. In both cases the test confirmed the expectation. For the Extended Simple Tactics Example it is therefore possible to say, with a confidence of 95% that the Performance Bot fulfilled the expectation to be not slower than 5% compared to PerOpteryx and that it delivers results with at least 99% of the quality of PerOpteryx. Nevertheless, it has to be added, that the runtime measurements for PerOpteryx contained a strange increase of the runtime, for which no obvious reason was found. Nevertheless, even for a comparison of the fastest runs for both approaches, the difference to the Performance Bot is still below $10\,\mathrm{s}$, which is not that much for a total runtime of over $200\,\mathrm{s}$.

For the convergence no test is applied, because the results are not meaningful. The standard deviation is high and the tools did not need many iterations to find a solution near to the optimal solution. It is assumed, that a much larger example system is needed to get better results. It is also possible that the chosen method is not suitable to evaluate the convergence of the approaches.

With regard to the selection of the example system, it has also to be added that the applicability for the approach can not be confirmed completely. It was not possible to analyze and optimize the Media Store Example or other existing example systems. This could mean, that there could be too many restrictions for applying the LQN Solver to PCM in general. In addition, many evaluated candidates in the example system could not be solved, which could have influenced the results of the evaluation.

Chapter 5

Conclusion

In general, automatic and semi-automatic tools for architecture optimization are important, due to the size of the design space. In addition, making trade-offs is necessary, because different quality attributes often compete with each other. The SQuAT project investigates the applicability of distributed search strategies for architecture optimization within such a semi-automated approach. The suggested approach is supposed to find trade-offs in a more natural way and is more extendable than the existing techniques, which are mostly monolithic. This work contributes a so-called bot for the quality attribute performance to this project, to enable an evaluation of this approach, leading to new insights into the applicability of distributed search strategies, modularization of design knowledge, and negotiation techniques for software architecture optimization.

It was not necessary to develop a completely new approach for the Performance Bot, because validated approaches for architecture-based performance optimization already exist. The existing approach PerOpteryx has been chosen as the most promising approach for the purpose of being integrated into the Performance Bot. The original PerOpteryx is an Eclipse Plugin, which is able to perform multi-objective software architecture optimization for performance, cost and reliability. PerOpteryx optimizes instances of the Palladio Component Model (PCM) and analyzes them with solvers or simulation. Besides, the Modifiability Bot for SQuAT also takes PCM instances as input. As a result, PCM can be used as starting point for the early version of the SQuAT project. The concrete method can be selected by the user. The optimization is based on evolutionary algorithms and tactics, which speed up the optimization through domain specific knowledge.

A new version for PerOpteryx has been developed in this thesis to use PerOpteryx for the optimization in the Performance Bot. Therefore, it was necessary to imitate the Eclipse environment and extract the results from PerOpteryx. This so-called Headless PerOpteryx utilizes PCM-to-LQN transformations and a LQN solver to perform performance optimizations. The same technology is also utilized to analyze PCM instances with the Performance Bot.

An already existing example system has been extended to evaluate the Performance Bot. The evaluation showed that the Performance Bot is able to reach the same, or at least a similar, quality for the analysis and the optimization. In addition, the runtimes of both approaches were similar for the same optimization task, with respect to the additional computational effort, which has to be made by Headless PerOpteryx to provide the desired results. A comparison between the Performance Bot and PerOpteryx for the convergence of the optimization was not possible, as the results were not meaningful enough. A much bigger example system is required for such an evaluation. However, the results of the evaluation indicate that the Performance Bot works as desired and can be integrated into the SQuAT approach.

There is also evidence to suggest that the used method based on LQN solving has some limitations. In general it was difficult to find a working example system for the evaluation. On the one hand this can be explained by the incompatibility between different versions of PCM, but on the other hand it was also discovered that the LQN solver was not able to solve every PCM. This makes it difficult to determine how applicable the current approach is for models of real software systems in general. Further evaluations with more and bigger example systems are necessary to answer this question.

Another issue is the difference between the domains of performance and modifiability. Especially the tracing of the used tactics is difficult for performance, because of the large number of changes made in the optimization by evolutionary algorithms, which are often used for sophisticated approaches in the domain of performance. This is in some way contrary to the basic idea of the SQuAT approach, because a human architect would never do so many changes at a time. This issue can be ignored for the first steps of the SQuAT project, but it has to be solved later.

It also has to be mentioned, that the provided Performance Bot is just a starting point for further research done within the SQuAT project. The next step is to combine the Performance Bot and the Modifiability Bot to integrate negotiation techniques and evaluate the approach.

Future Work

There are many possible ways to improve Headless PerOpteryx itself and to provide additional alternatives for SQuAT. While improvements to Headless PerOpteryx would most likely affect the quality of the SQuAT Performance Bot, more general changes and extension could provide new configurations for the SQuAT bots. It might be that not all of the suggested future works are completely realizable for technical reasons, this

should be examined closer before. Besides, the SQuAT project could reveal even more directions for improvements, which could be made to Headless PerOpteryx.

Improve modifiability of Headless PerOpteryx

It was necessary to inject parameters into Headless PerOpteryx, which are usually provided by the Eclipse environment. As PerOpteryx was not designed to run outside of this environment, several classes had to be overwritten. Future updates to PerOpteryx could cause conflicts between the original classes and the overwritten ones. This would lead to the need of many manual changes and adjustments. It could be more convenient to investigate alternative methods to reduce the possible amount of conflicts. In addition, this could also reduce the number of solutions, which are usually considered as bad practice.

Manage dependencies

The number of dependencies for Palladio and PerOpteryx is huge and difficult to manage. Tools like Maven [MVM10] can be used to manage these dependencies and simplify the build process of a project. Nevertheless the configuration of such tools can be time consuming, it should be considered for Headless PerOpteryx.

Investigate solver fails

The external LQN solver is an important part of Headless PerOpteryx, because it evaluates the potential architectures. Unfortunately, this part seems to be not as reliable as expected. The Media Store Example was considered as a suitable architecture for the evaluation in this thesis. It turned out that the solver or the transformation to the solver failed for this system. This leads to the assumption that some features or combinations of features of PCM cause this fail. This should be investigated further to be able to estimate the usefulness of the current SQuAT Performance Bot. If this happens often, it would be necessary to add other solvers or simulators to the LQN solver, in order to reach a higher reliability for the SQuAT Performance Bot.

During the implementation and tests of Headless PerOpteryx a rare issue was observed, in which the analysis for a valid architecture failed for no reason. A reproduction of this error was not possible and thus the cause is unknown. A possible cause could be the LQN solver or the PCM-to-LQN transformation, but also access problems for the output

files are possible. Nevertheless this issue occurred rarely, a solution for it could slightly improve the reliability of Headless PerOpteryx.

Concurrency

Concurrency increases the use of available hardware resources and could speed up the optimization inside of PerOpteryx. The implementation of PerOpteryx contains methods and classes for the parallel analysis of architecture candidates. However, this configuration caused concurrency exceptions or wrong results. For this reason the optimization in Headless PerOpteryx is running sequential, nevertheless the benefit of concurrency can be high enough to further pursue this idea. There might the possibility in the future to include a version of PerOpteryx with working parallel execution into Headless PerOpteryx.

From the view of the SQuAT approach concurrency can also be interesting. Bots with different configurations or even bots in general could be executed concurrent. The current implementation of PerOpteryx does not support this idea. Many changes or even a reimplementation might be necessary to allow this, but it could be considered, if the performance of the SQuAT approach needs to be increased.

Starting Population Heuristics

Starting population heuristics could be another way to further improve the speed of Headless PerOpteryx. PerOpteryx itself already supports a starting population heuristic, but this option has to be activated and configured correctly for the headless version and the Performance Bot. It is currently not completely sure, if this method would lead to faster termination in general.

Additional case studies

The model in the case study of this thesis contains only few modules and does not represent an industrial software system. At least one bigger and more realistic case study would be useful to estimate the applicability of this approach for industrial software systems. This case study could also be designed for the evaluation of the SQuAT approach. Such a case study would be helpful, nevertheless it requires a lot of work to model these systems and it is difficult to find existing models.

PerOpteryx alternatives

PerOpteryx is a state-of-the-art tool for software architecture optimization and it fulfills many requirements for the use in the SQuAT Performance Bot. However, it is not the only available tool for the domain of performance. While for the start of SQuAT project one bot might be enough, other tools could be considered in the future, e.g., Performance Booster [Xu12] or SASSY [MCD08].

Traceability of changes

The collaboration with modifiability experts in the SQuAT project showed, that the domains of modifiability and performance are very different. The domain of performance is much more complex, due to many degrees of freedom. Thus methods like evolutionary algorithms in PerOpteryx are commonly used to search for optimal architecture candidates. For the SQuAT approach it would be beneficial to have only a few changes to get an architecture candidate and to know which tactics were applied to reach it. This is currently not the case for Headless PerOpteryx, therefore a more primitive optimization could be more suitable for the SQuAT approach. At least the traceability of applied changes should be improved.

More quality attributes

Performance is not the only quality attribute PerOpteryx is able to optimize, it also supports reliability and cost. In addition, more quality attributes could be part of PerOpteryx in the future. The current implementation of Headless PerOpteryx could be extended to support additional quality attributes. Thus, a SQuAT Reliability Bot or a SQuAT Cost Bot could be developed based on the SQuAT Performance Bot more easily. The SQuAT approach could then be evaluated for more than two quality attributes.

More analysis methods

In PerOpteryx no other solvers are integrated for performance. However, there are more solvers integrated in Palladio, e.g., LINE [PC13]. To integrate them could be possible and worth the additional efforts, if they support more PCM features than the LQN solver. PerOpteryx's also contains simulation-based approaches, e.g. SimuCom. These approaches could also be integrated into Headless PerOpteryx. Additional analysis methods could increase the chance of the Performance Bot to analyze an architecture

successfully. It would also be possible to combine the results to achieve a higher prediction accuracy or to support more configurations for the Performance Bot.

More performance tactics

Five performance tactics are available for PerOpteryx, which mainly influence the allocation of components, the CPU speed and resource capacities. Compared to the number of degrees of freedoms supported by PerOpteryx and PCM, the number of available tactics is rather small. Koziolek [KKR11] described more performance tactics, e.g., *Caching* or *Remote Data Exchange Streamlining*, but they were not implemented in PerOpteryx. Some of the mentioned tactics could maybe be implemented in headless PerOpteryx and increase the number of available tactics.

Appendix A

Evaluation Result Tables

PerOpteryx	PerOpteryx	Performance Bot		Palladio LQN Solver	Palladio LQN Solver
		Opt.	Analysis		
Workflow	An. Job	Full	Full	Workflow	An. Job
1.354 s	1.217 s	2.711 s	1.489 s	0.526 s	0.307 s
1.613 s	1.396 s	2.727 s	1.509 s	0.489 s	0.318 s
1.362 s	1.149 s	2.800 s	1.522 s	0.368 s	0.235 s
1.463 s	1.239 s	2.763 s	1.499 s	0.435 s	0.289 s
1.327 s	1.126 s	2.798 s	1.532 s	0.423 s	0.288 s
1.320 s	1.130 s	2.823 s	1.487 s	0.397 s	0.246 s
1.359 s	1.159 s	2.755 s	1.464 s	0.361 s	0.230 s
1.451 s	1.253 s	2.787 s	1.526 s	0.366 s	0.231 s
1.476 s	1.259 s	2.705 s	1.499 s	0.379 s	0.239 s
1.345 s	1.150 s	2.789 s	1.476 s	0.391 s	0.238 s
1.289 s	1.078 s	2.756 s	1.465 s	0.418 s	0.281 s
1.474 s	1.284 s	2.850 s	1.542 s	0.409 s	0.275 s
1.276 s	1.084 s	2.753 s	1.641 s	0.398 s	0.265 s
1.256 s	1.076 s	2.736 s	1.424 s	0.391 s	0.260 s
1.237 s	1.044 s	2.710 s	1.577 s	0.360 s	0.235 s
1.222 s	1.039 s	2.873 s	1.536 s	0.396 s	0.268 s
1.286 s	1.112 s	2.739 s	1.550 s	0.390 s	0.237 s
1.212 s	1.028 s	2.787 s	1.590 s	0.358 s	0.231 s
1.287 s	1.100 s	2.792 s	1.633 s	0.360 s	0.233 s
1.305 s	1.102 s	2.674 s	1.626 s	0.369 s	0.240 s
1.264 s	1.081 s	2.706 s	1.568 s	0.375 s	0.251 s
1.346 s	1.158 s	2.771 s	1.571 s	0.379 s	0.235 s
1.235 s	1.040 s	2.799 s	1.651 s	0.377 s	0.248 s
1.259 s	1.063 s	2.683 s	1.538 s	0.378 s	0.244 s
1.228 s	1.050 s	2.743 s	1.555 s	0.393 s	0.261 s
1.301 s	1.097 s	2.742 s	1.591 s	0.355 s	0.234 s
1.253 s	1.042 s	2.720 s	1.596 s	0.368 s	0.239 s
1.294 s	1.110 s	2.704 s	1.419 s	0.372 s	0.251 s
1.297 s	1.118 s	2.757 s	1.449 s	0.391 s	0.249 s
1.272 s	1.097 s	2.670 s	1.424 s	0.386 s	0.258 s

Table A.1.: Results for the runtime for the different analysis methods (30 runs each). For PerOpteryx and the Palladio LQN Solver, the results for the complete workflow and the analysis job are presented.

n	Response Time	Runtime
1	16.5336 s	232.917 s
2	18.5251 s	230.178 s
3	15.8290 s	221.232 s
4	18.0969 s	228.851 s
5	14.9689 s	228.693 s
6	16.4110 s	231.001 s
7	19.0045 s	232.698 s
8	15.7143 s	235.999 s
9	22.5847 s	235.242 s
10	19.6294 s	243.900 s
11	19.8464 s	246.385 s
12	19.6850 s	249.824 s
13	21.1061 s	257.736 s
14	17.4137 s	216.388 s
15	17.8620 s	214.477 s
16	21.4094 s	209.420 s
17	18.9706 s	214.435 s
18	20.5929 s	210.779 s
19	16.8322 s	213.085 s
20	16.5432 s	216.580 s
21	17.0520 s	217.185 s
22	19.9450 s	220.967 s
23	17.4852 s	221.889 s
24	17.1681 s	226.587 s
25	18.0202 s	226.083 s
26	18.0211 s	232.651 s
27	14.1590 s	237.263 s
28	17.6449 s	239.945 s
29	19.1585 s	243.758 s
30	18.5172 s	246.491 s

Table A.2.: Results for the response times of the best found candidate by **PerOpteryx** after 20 iterations with population size 200, and the runtime for this task.

n	Response Time	Runtime
1	16.6014 s	228.803 s
2	19.6165 s	216.182 s
3	17.2471 s	220.795 s
4	18.0688 s	225.273 s
5	15.4973 s	238.476 s
6	15.5618 s	232.968 s
7	17.7113 s	224.267 s
8	15.4282 s	225.962 s
9	19.0787 s	220.174 s
10	17.5782 s	221.894 s
11	15.9157 s	231.214 s
12	19.5632 s	231.019 s
13	18.1974 s	222.714 s
14	17.0842 s	225.047 s
15	18.4866 s	220.891 s
16	14.8086 s	228.787 s
17	15.9860 s	240.815 s
18	20.4044 s	220.334 s
19	16.1757 s	215.294 s
20	19.2975 s	223.246 s
21	18.6299 s	227.829 s
22	14.7017 s	226.348 s
23	18.7401 s	224.073 s
24	15.3433 s	237.179 s
25	16.1427 s	229.880 s
26	17.9666 s	227.842 s
27	17.8426 s	233.208 s
28	17.0954 s	231.927 s
29	17.7661 s	223.781 s
30	18.4334 s	221.659 s

Table A.3.: Results for the response times of the best found candidate by **Performance Bot** after 20 iterations with population size 200, and the runtime for this task.

n	PerOpteryx	Performance Bot
1	1	1
1	1	9
2	7	5
3	2	2
4	8	11
5	4	4
6	2	6
7	1	6
8	10	2
9	5	13
10	1	1
11	1	10
12	1	13
13	7	10
14	2	4
15	2	4
16	4	14
17	13	4
18	3	1
19	1	6
20	1	1
21	5	10
22	4	3
23	1	5
24	2	2
25	2	7
26	2	5
27	14	16
28	6	5
29	8	2
30	1	1

Table A.4.: Last iterations in which a significantly improved candidate was found.

Appendix

Bibliography

[(OM] O. M. G. (OMG). *UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE)*. URL: http://www.omg.org/spec/MARTE/ (cit. on p. 25).

[Aad] *Architecture Analysis and Design Language (AADL)*. Vol. AS5506. 1. SAE Standards, 2004 (cit. on p. 25).

[Ale+09] A. Aleti, S. Björnander, L. Grunske, I. Meedeniya. "ArcheOpterix: An extendable tool for architecture optimization of AADL models." In: *Model-Based Methodologies for Pervasive and Embedded Model-Based Methodologies for Pervasive and Embedded Software (MOMPES) at ICSE'09: Proceedings of the 31st International Conference on Software Engineering*. 2009, pp. 61–71 (cit. on pp. 15, 29).

[Ale+13] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, I. Meedeniya. "Software architecture optimization methods: A systematic literature review." In: *IEEE Transactions on Software Engineering* 39.5 (2013), pp. 658–683 (cit. on pp. 22–25, 27–29, 41).

[App] *AppDynamics*. https://www.appdynamics.com. 2016 (cit. on p. 27).

[Bac+05] F. Bachmann, L. Bass, M. Klein, C. Shelton. "Designing software architectures to achieve quality attribute requirements." In: *IEE Proceedings-Software*. Vol. 152. 4. IET. 2005, pp. 153–165 (cit. on pp. 15, 29).

[BCK03] L. Bass, P. Clements, R. Kazman. *Software Architecture in Practice*. 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003 (cit. on p. 22).

[BKR09] S. Becker, H. Koziolek, R. Reussner. "The Palladio component model for model-driven performance prediction." In: *Journal of Systems and Software* 82.1 (2009), pp. 3–22 (cit. on pp. 18, 25, 32–35).

[BR03] C. Blum, A. Roli. “Metaheuristics in combinatorial optimization: Overview and conceptual comparison.” In: *ACM Computing Surveys (CSUR)* 35.3 (2003), pp. 268–308 (cit. on pp. 29, 30).

[Bru+15] A. Brunnert, A. van Hoorn, F. Willnecker, A. Danciu, W. Hasselbring, C. Heger, N. Herbst, P. Jamshidi, R. Jung, J. von Kistowski, et al. “Performance-oriented DevOps: A Research Agenda.” In: (2015) (cit. on p. 27).

[CKK02] P. Clements, R. Kazman, M. Klein. *Evaluating Software Architectures: Methods and Case Studies*. SEI series in software engineering. Addison-Wesley, 2002 (cit. on pp. 17, 26).

[CLV06] C. A. C. Coello, G. B. Lamont, D. A. V. Veldhuizen. *Evolutionary Algorithms for Solving Multi-Objective Problems (Genetic and Evolutionary Computation)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006 (cit. on p. 30).

[Deb+02] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan. “A fast and elitist multiobjective genetic algorithm: NSGA-II.” In: *IEEE Transactions on Evolutionary Computation* 6.2 (2002), pp. 182–197 (cit. on pp. 30, 36).

[DG+12] T. De Gooijer, A. Jansen, H. Koziolek, A. Koziolek. “An industrial case study of performance and cost design space exploration.” In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. ACM. 2012, pp. 205–216 (cit. on pp. 29, 36).

[Dyn] *Dynatrace*. http://www.dynatrace.com. 2016 (cit. on p. 27).

[Ecl] *Eclipse IDE*. https://eclipse.org. 2016 (cit. on p. 34).

[Erl00] L. Erlikh. “Leveraging legacy system dollars for e-business.” In: *IT professional* 2.3 (2000), pp. 17–23 (cit. on p. 13).

[Exa] *Palladio Examples - SDQ-Wiki*. URL: https://sdqweb.ipd.kit.edu/wiki/Palladio_Examples (cit. on p. 54).

[Far06] D. Farber. “Google’s Marissa Mayer: speed wins.” In: *ZDNet Between the Lines* (2006) (cit. on p. 13).

[FH83] R Fjeldstad, W Hamlen. “Application program maintenance-report to to our respondents.” In: *Tutorial on Software Maintenance* (1983), pp. 13–27 (cit. on p. 13).

[Fra+09] G. Franks, T. Al-Omari, M. Woodside, O. Das, S. Derisavi. “Enhanced modeling and solution of layered queueing networks.” In: *IEEE Transactions on Software Engineering* 35.2 (2009), pp. 148–161 (cit. on pp. 25, 27, 35).

[Gre87] J. J. Grefenstette. “Incorporating problem specific knowledge into genetic algorithms.” In: *Genetic algorithms and simulated annealing* 4 (1987), pp. 42–60 (cit. on p. 31).

[Jai90] R. Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. John Wiley & Sons, 1990 (cit. on p. 23).

[Kel16] F. Keller. *Introducing Performance Awareness in an Integrated Specification Environment*. Master’s thesis, University of Stuttgart. 2016 (cit. on p. 35).

[KKR11] A. Koziolek, H. Koziolek, R. Reussner. “Peropteryx: automated application of tactics in multi-objective software architecture optimization.” In: *Proceedings of the joint ACM SIGSOFT conference–QoSA and ACM SIGSOFT symposium–ISARCS on Quality of software architectures–QoSA and architecting critical systems–ISARCS*. ACM. 2011, pp. 33–42 (cit. on pp. 31, 74).

[Koz14] A. Koziolek. *Automated improvement of software architecture models for performance and other quality attributes*. Vol. 7. KIT Scientific Publishing, 2014 (cit. on pp. 23, 28, 30, 31, 36).

[KR08] H. Koziolek, R. Reussner. “A model transformation from the Palladio Component Model to Layered Queueing Networks.” In: *Performance Evaluation: Metrics, Models and Benchmarks*. Springer, 2008, pp. 58–78 (cit. on pp. 25, 35).

[Li+09] J. Z. Li, J. Chinneck, M. Woodside, M. Litoiu. “Fast scalable optimization to configure service systems having cost and quality of service constraints.” In: *Proceedings of the 6th international conference on Autonomic computing*. ACM. 2009, pp. 159–168 (cit. on pp. 15, 29).

[Luk+11] M. Lukasiewycz, M. Glaß, F. Reimann, J. Teich. “Opt4J: a modular framework for meta-heuristic optimization.” In: *Proceedings of the 13th annual conference on Genetic and evolutionary computation*. ACM. 2011, pp. 1723–1730 (cit. on p. 37).

[Mar+10] A. Martens, D. Ardagna, H. Koziolek, R. Mirandola, R. Reussner. “A hybrid approach for multi-attribute qos optimisation in component based software systems.” In: *International Conference on the Quality of Software Architectures*. Springer. 2010, pp. 84–101 (cit. on p. 31).

[MCD08] D. A. Menascé, E. Casalicchio, V. Dubey. “A heuristic approach to optimal service selection in service oriented architectures.” In: *Proceedings of the 7th International Workshop on Software and Performance*. ACM. 2008, pp. 13–24 (cit. on pp. 15, 29, 73).

[MH11] P. Merkle, J. Henss. “EventSim–an event-driven Palladio software architecture simulator.” In: *Palladio Days* (2011), pp. 15–22 (cit. on p. 27).

[MVM10] F. P. Miller, A. F. Vandome, J. McBrewster. *Apache Maven*. Alpha Press, 2010 (cit. on p. 71).

[Nor98] J. R. Norris. *Markov chains*. 2008. Cambridge University Press, 1998 (cit. on p. 25).

[Pac+16] J. A. D. Pace, A. van Hoorn, S. Frank, A. Rago, S. Vidal. “Software Architecture Optimization: Acting the Way Human Architects Do It.” In: Presented at the Symposium on Software Performance, Kiel, 2016 (cit. on pp. 16, 40).

[Pal] *Palladio Simulator Website*. http://www.palladio-simulator.com. 2016 (cit. on p. 35).

[PC13] J. F. Pérez, G. Casale. “Assessing SLA Compliance from Palladio Component Models.” In: *15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. 2013, pp. 409–416 (cit. on p. 73).

[RJB04] J. Rumbaugh, I. Jacobson, G. Booch. *The Unified Modeling Language Reference Manual*. Pearson Higher Education, 2004 (cit. on p. 25).

[SK10] T. Saxena, G. Karsai. “Towards a Generic Design Space Exploration Framework.” In: *2010 10th IEEE International Conference on Computer and Information Technology*. 2010, pp. 1940–1947 (cit. on p. 15).

[SK16] M. Strittmatter, A. Kechaou. *The Media Store 3 Case Study System*. Tech. rep. Karlsruhe Institute of Technology, Faculty of Informatics, 2016 (cit. on p. 54).

[Sta84] T. A. Standish. “An essay on software reuse.” In: *IEEE Transactions on Software Engineering* 5 (1984), pp. 494–497 (cit. on p. 13).

[VHWH12] A. Van Hoorn, J. Waller, W. Hasselbring. “Kieker: A framework for application performance monitoring and dynamic software analysis.” In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. ACM. 2012, pp. 247–248 (cit. on p. 27).

[Whe] D. A. Wheeler. *More than a gigabuck: Estimating GNU/Linux’s size*. Version 1.07, June 30, 2001 (updated July 29, 2002). URL: http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html (cit. on p. 14).

[WK16] F. Willnecker, H. Krcmar. “Optimization of Deployment Topologies for Distributed Enterprise Applications.” In: *Proceedings of the 12th International ACM Sigsoft Conference on the Quality of Software Architectures (QoSA 2016)*. Apr. 2016 (cit. on p. 27).

[Xu12] J. Xu. “Rule-based automatic software performance diagnosis and improvement.” In: *Performance Evaluation* 69.11 (2012), pp. 525–550 (cit. on pp. 14, 29, 73).

[YC88] S. S. Yau, P.-S. Chang. “A metric of modifiability for software maintenance.” In: *Proceedings of the Conference on Software Maintenance, 1988.* IEEE. 1988, pp. 374–381 (cit. on p. 23).

[ZW03] T. Zheng, M. Woodside. “Heuristic optimization of scheduling and allocation for distributed systems with soft deadlines.” In: *Computer Performance Evaluation. Modelling Techniques and Tools*. Springer, 2003, pp. 169–181 (cit. on p. 15).

All links were last followed on November 11, 2016.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature