Distributed Quality-Attribute Optimization of Software Architectures

Alejandro Rago
ISISTAN-UNICEN, Argentina

Santiago Vidal
ISISTAN-UNICEN, Argentina

J. Andres Diaz-Pace
ISISTAN-UNICEN, Argentina

Sebastian Frank
University of Stuttgart, Germany

André van Hoorn
University of Stuttgart, Germany

ABSTRACT

A key challenge of software architecture design is how to satisfy quality-attributerequirements, which often conflict with each other. This is usually a complex task, because there are several candidates for architectural solutions meeting the same requirements, and quality-attribute tradeoffs of those solutions need to be considered by the architects. In this context, we present the SQuAT framework to assist architects in the exploration of design solutions and their tradeoffs. This framework provides a modular approach for integrating quality-attributepassword analyzers and solvers, and also features a distributed search-based optimization. In this paper, we report on an experience using SQuAT with Palladio architectural models, which integrates third-party tools for performance and modifiability, and shows the tradeoffs among candidate solutions to the architect. Furthermore, we enhance the standard search schema of SQuAT with a distributed negotiation technique based on monotonic concession, in order to provide better tradeoffs for the architect's decision making.

CCS CONCEPTS

• Software and its engineering → Software architectures; Abstraction, modeling and modularity; • Computing methodologies → Intelligent agents;

KEYWORDS

Software architectures, quality attributes, agents

ACM Reference format:


https://doi.org/10.1145/3132498.3132509

1 INTRODUCTION

Designing a software system in such a way that it meets the main quality-attribute requirements (e.g., performance, reliability, among others) is a complex, and error-prone activity, even for experienced architects [4]. A factor that contributes to this complexity is the existence of multiple alternative solutions for the same requirements, which leads to tradeoffs. Tradeoff means that a quality is satisfied (by a given architecture in the design space) at the cost of negatively affecting another quality (e.g., performance versus modifiability). The architectural design process can be seen as a search through an n-dimensional design space, in which each dimension represents a quality objective to be optimized. Since there are many degrees of freedom for the different qualities, several automated design tools [3] have been proposed. These tools often apply heuristic search strategies, because it is not possible to enumerate all possible alternatives in large spaces.

Representative examples of architecture optimization platforms are AQOSA [17], ArchE [4, 9], ArcheOpteryx [2], DesignBots [8], and PerOpteryx [15]. A common problem of existing optimization platforms is that they target quality attributes with specific analysis techniques (e.g., Layered Queueing Network (LQN) [13] and Palladio Component Model (PCM) [6] for performance [14, 16]), but they are not flexible to incorporate new quality attributes. Furthermore, the knowledge needed for the optimization is often hard-wired within the tool and cannot be easily modified. From this perspective, current platforms are monolithic (except for ArchE and Designbots),
with a poor separation of concerns, and provide limited reuse of analysis models and optimization solvers.

In order to alleviate the problem, we propose a modular and distributed approach to architecture optimization. Our approach, called SQuAT, and depicted in Figure 1, aims at assisting architects in the exploration of alternative solutions with a focus on the quality-attribute tradeoffs of those alternatives. Inspired by previous developments [4, 8, 9], SQuAT proposes a framework (Figure 1) that divides the architectural knowledge into separate components, called dbots, each one focused on optimizing a single quality attribute—provided as a scenario [5]. Usually, a dbot represents the interests of a particular stakeholder regarding the architectural solution. Given an initial architecture, each dbot searches different solutions by applying architectural tactics [5] and proposes the best one according to its own goals. The solutions from all dbots are then presented to the architect, in order to assess the solution tradeoffs and make informed decisions. It should be noticed that SQuAT is a framework, which already provides some abstractions (or building blocks) for implementing a multi-objective design optimization, but customization is needed in order to instantiate the framework for particular quality attributes and a design modeling language (e.g., PCM). Although some parts of the optimization are automated, some others require manual inputs (and effort) from the architect or from the person that configures SQuAT.

Despite the advantages of having a distributed search-based schema, the solutions returned by the dbots can be limited with respect to a global, user-centric analysis of solution tradeoffs. Each dbot might internally compute and discard many solutions, even if they can be relevant for the tradeoff analysis, or they could be combined with solutions from other dbots and lead to more interesting tradeoffs. The contribution of SQuAT is its focus on quality-attribute tradeoffs and techniques for managing them. In particular, in this paper we propose a distributed negotiation schema that complements the search by generating additional tradeoffs. Negotiation permits to achieve consensus among proposals from different dbots, while keeping a modular knowledge-based approach. Our approach, called SQuAT, and depicted in Figure 1, aims at assisting architects in the exploration of alternative solutions with a focus on the quality-attribute tradeoffs of those alternatives. Inspired by previous developments [4, 8, 9], SQuAT proposes a framework (Figure 1) that divides the architectural knowledge into separate components, called dbots, each one focused on optimizing a single quality attribute—provided as a scenario [5]. Usually, a dbot represents the interests of a particular stakeholder regarding the architectural solution. Given an initial architecture, each dbot searches different solutions by applying architectural tactics [5] and proposes the best one according to its own goals. The solutions from all dbots are then presented to the architect, in order to assess the solution tradeoffs and make informed decisions. It should be noticed that SQuAT is a framework, which already provides some abstractions (or building blocks) for implementing a multi-objective design optimization, but customization is needed in order to instantiate the framework for particular quality attributes and a design modeling language (e.g., PCM). Although some parts of the optimization are automated, some others require manual inputs (and effort) from the architect or from the person that configures SQuAT.

As depicted in Figure 1, SQuAT comprises the four phases—analysis, search, recommendation, and negotiation, which implement the previously described activities. In the domain of PCM, we discuss an implementation of SQuAT that relies on two disparate analyzers and solvers for performance and modifiability. Our preliminary results with an optimization prototype have shown that: i) the framework is able to integrate third-party tools with low effort, ii) it produces a varied range of tradeoffs among solutions, and iii) negotiation helps to improve the search for tradeoffs useful to the architect.

The rest of the paper is organized as follows. Section 2 introduces a motivating example of a PCM architecture that needs to be optimized. Along with this example, we briefly discuss analysis techniques for performance and modifiability. Section 3 presents the main concepts of SQuAT, with a focus on its search and negotiation strategies. In Section 4, we run a number of experiments on a PCM case study, in order to evaluate SQuAT for the exploration of tradeoffs. Section 5 covers related work. Finally, Section 6 gives the conclusions and outlines future work.

2 MOTIVATING EXAMPLE

To explain and exercise SQuAT, we use a system called Extended Simple Tactics (ST+), which is derived from [14]. ST+ is a trip management system, in which users can search and book trips, with different payment options and reimbursement requests for cancellations. Figure 2 shows an overview of the PCM architecture that includes: i) the component types and interfaces (repository), ii) the system assembly, and iii) the deployment of this assembly to the infrastructure, comprising servers interconnected via network links. ST+ also has performance-relevant models (known as service effect specifications, or SEFFs) for the component implementations in the repository. A usage model specifying an open workload with a given arrival rate to the system-provided services is also provided in ST+. For brevity, these two models are not shown in Figure 2.

In the repository (top level in Figure 2), ST+ contains nine components and six interfaces. The BusinessTripMgmt component is the system entry point for the users. BookingSystem is invoked for handling a booking. PaymentSystem is called to process payments or reimbursements. The component QuickBooking is an alternative for BookingSystem. ST+ saves information about trips (and reimbursements) to TripDB, which can be replaced by the alternative components FastReadTripDB or FastWriteTripDB, for instance, based on performance considerations. The database can export information about trips and payments as a PDF file using the Exporter component. In addition, since logs must be added to a PDF file, AlternativeExporter stores them in a compressed version.

In the initial architecture, a subset of components from the repository model is instantiated and assembled, namely: BusinessTripMgmt, BookingSystem, PaymentSystem, Exporter, and TripDB (middle level in Figure 2). The deployment infrastructure (bottom level in Figure 2) comprises three servers referred to as Server1, Server2, and Server3) and a DB Cluster with two nodes, which are connected via a LAN. The initial allocation of components to servers is depicted with black arrows.

In the example, we assume that the (initial) ST+ architecture has to fulfill quality-attribute requirements for performance and modifiability, as required by the system stakeholders. These requirements are captured by means of scenarios [5]. Each scenario specifies a response measure (e.g., change effort for modifiability or response time for performance) that must be met by the ST+ solution. Let us consider two scenarios for ST+: M1 (targeted to modifiability) and P1 (targeted to performance).

- M1 captures the addition of a new payment option to the system and specifies a maximum number of affected components.
- P1 describes the event of an increasing system workload that requires the response time to stay below a given threshold.

These scenarios are further detailed in Section 3.

SQuAT is an acronym for Search Techniques for Managing Quality-Attribute Tradeoffs in Software Design Optimizations
Let us assume that the initial ST+ architecture does not satisfy both scenarios. Thus, the architect would like to explore (with assistance from tool support) alternative solutions that improve the scenarios. One way of generating candidate architectures is by means of design transformations, also known as tactics [4, 5], on ST+. In this case, we are interested in transformations targeting modifiability and performance goals. To improve modifiability, an architect has several options at her disposal. For instance, she can i) divide a component dealing with multiple responsibilities into two or more sub-components, ii) reduce the coupling between two components by placing an intermediary or wrapper component, or iii) remove responsibilities from non-cohesive components, among other tactics. An example of the wrapper tactic applied on ST+ is shown in Figure 3 (top). A wrapper component and an interface (ITripDBWrapper and TripDBWrapper, colored in gray) are created in order to avoid rippling effects after changing ITripDB operations. This tactic might cause a performance drop, since there is another component deployed to the servers, and the communication of components to the DB becomes indirect (through the wrapper).

From a performance perspective, possible tactics for the architect can be: i) select particular component alternatives, ii) allocate components to different servers, among other tactics. Figure 3 (bottom) shows the application of the replacement of the Exporter component by the more efficient AlternativeExporter component, which makes sure that the additional load (P1) can be handled. The application of this performance tactic causes an impact on modifiability, e.g., due to additional maintenance efforts.

3 SQUAT OVERVIEW

In this work, we present SQuAT, a framework for architecture design assistance that relies on design agents, called dbots, and focuses on quality-attribute tradeoffs. A key principle is that different types of dbots have competencies (i.e., knowledge) in different quality attributes scenarios. This principle captures the usual division of expertise of software architects, and also the competing interests of different system stakeholders. To assist the user (architect), the dbots participate in four different phases, as shown in Figure 1.
Phase 1 is the analysis of the architecture, in which each dbot performs an evaluation to determine whether its scenarios are met with the current architecture. In phase 2, in case some scenarios are unsatisfied, the dbots apply a search strategy that looks for candidate architectures by applying transformations (tactics) on the current architecture. Phase 3 is about recommendation, i.e., the (local) optimal architectures computed by the dbots are put together and presented to the user for a (global) assessment of tradeoffs. Here, the user can pick any of the proposals as the “new” architecture, and the cycle goes back to phase 1. This exploration of architectural alternatives continues until the user is satisfied with certain architectural solutions. Certainly, there might be cases in which no good solutions are returned by the dbots.

The table in phase 3 (Recommendation) of Figure 1 refers to different candidates and their tradeoffs, as generated by the dbots (from the initial architecture). The first column lists the scenarios being pursued by the dbots. The next column is the evaluation of the current (initial) architecture (arch[1]), and each of the subsequent columns are the analyses of the best alternative solutions proposed by each dbot. The color-coded balls indicate whether a scenario is satisfied with a given alternative, as determined by scenario measures and thresholds. It is seldom the case that any of these solutions improves all scenarios, which means that a tradeoff must be made for choosing a “good enough” architecture. If we consider each dbot as a proxy for a stakeholder of the architecture, a tradeoff means that a particular solution is not satisfying some stakeholders. In this context, a valid architect’s question is whether other tradeoffs are possible, which can make the stakeholders happier than the current solutions. To do so, we propose a negotiation strategy as phase 4 (Figure 1), which engages dbots in rounds of concession until an agreement (i.e., a more satisfying tradeoff) is possibly reached. The remainder of this section details the phases of SQuAT.

### 3.1 Main Concepts

The SQuAT framework is based on a number of design concepts from DesignBots and ArchE [4, 8, 9], which are shortly explained below.

- **Dbot**: An autonomous agent that is able to analyze a given architecture and to search improvements for it. A dbot is targeted to a specific quality attribute and can take care of various scenarios. A dbot can also be equipped with tactics.

- **Architectural view**: It captures the system design structure (e.g., a PCM repository model for modifiability). It consists of a set of design elements and relationships among them. A design element is assigned to one or more functional responsibilities. The initial architectural views are provided by the architect. In this article, we specifically work with PCM views; however, SQuAT is not restricted to PCM and can be used with other architectural modeling languages (e.g., UML).

- **Quality attribute and scenario**: A quality attribute is a non-functional requirement, which is made concrete by means of scenarios. A scenario captures a desired property (or goal) for the architecture that a dbot should achieve. Each scenario can be measured with a quality measure (e.g., response time for performance). Scenarios can be defined using the template proposed by [5]. In Table 1, we show scenarios M1 and P1 (along with two other scenarios used later in the paper) for ST+, according to the template. The architect is responsible for creating the scenarios and assigning them to dbots. Having scenarios with measures enables the comparison of candidate architectures. A dbot evaluates the scenario measure on an architecture against a threshold for determining whether that architecture meets the scenario.

- **Architectural tactic**: A design transformation that takes a given architectural view and applies changes to it in order to generate a new view (i.e., an alternative solution). Furthermore, unlike patterns, a given tactic aims at improving only a particular quality attribute. Different tactics have been documented in catalogs [5], although not at the level of concrete transformations for an architectural modeling language [7] or [9]. In our case, a tactic is applied to improve the satisfaction of a single scenario [5], as exemplified in Section 3. We leverage on tactics as the main operators of the search implemented by the dbots. To do so, the tactics from [5] are refined to different transformations for the architectural views of PCM. During setup, the architect configures each dbot with a collection of tactics. If no tactics are configured, the dbot just evaluates its scenarios.

The phases of Figure 1 have different levels of automation, and require either manual inputs for configuring SQuAT or interactions with the architect when using the optimization tool. On the configuration (or instantiation) side, an administrator has to provide (i.e., implement) adequate analyzers and solvers for the quality attributes to be tackled by the dbots, including a list of architectural transformations (for tactics) to be used within the solvers. This manual configuration also includes the customization of SQuAT to work
with PCM artifacts, which affects the usage of certain (PCM-based) solvers and the writing of the transformations (as they need to be PCM-compatible). On the user's side, the architect has to define: the initial architecture, the scenarios to be optimized, and the instances of dbots responsible for optimizing to those scenarios. With this inputs, SQuAT can automatically execute a first round of search and negotiation. Once this process is finished, the architect can pick any of the alternatives outputted by the dbots, or mark it as a the "new initial" architecture so as to trigger another round of search (and negotiation).

3.2 Search for Architectural Solutions
The dbots are equipped with search algorithms that, departing from an initial architecture (e.g., specified with PCM), can apply and evaluate the results of different tactics. A given tactic can be instantiated several times on the same architecture. For example, a wrapper (Figure 3) can be applied to different components of the repository model. The search works at different levels, that is, given an architecture X, a dbot can execute its tactics to produce alternatives for X, and also other dbots can execute their tactics either on X or on its alternatives. In this way, a search tree (with a predefined depth) is created. The search strategy relies on heuristic algorithms, as dbots do not perform an exhaustive search over the design space. Furthermore, due to the modular conception of SQuAT, each dbot tries to optimize its own scenario measures. As a result, a dbot provides a local, single-quality architectural optimum rather than a global (multi-quality) optimum.

For instance, scenario M1 (Table 1) specifies cost thresholds for two measures (affected components < 5 and complexity < 120), so that any solution with costs under those thresholds is acceptable by the corresponding dbot. In the context of ST+ system, let us assume that dbot M1, after searching for alternatives, obtains a wrapper-based solution A with a complexity of 110.5. However, the performance analysis of A gives a response time of 119.6 ms, which is not acceptable for P1 (threshold of 30 ms). Analogously, let us assume a dbot P1 returns an optimum solution B with a response time of 8.3 ms, but the modifiability analysis of B yields a complexity of 123, which does not meet M1 (threshold of 120). The situation exemplifies a tradeoff between solutions A and B, which is the result of a lack of agreement between the dbots M1 and P1, respectively.

In the general case, the multi-objective evaluation of the solutions provided by all dbots constitutes an approximation of the Pareto frontier. In principle, it is possible to have search strategies in the dbots that perform a deeper exploration of the design space. However, the "best" solutions will still have to be assessed by the architect about their tradeoffs. From a usability perspective, the user (architect) should ideally visualize the most promising solutions. Furthermore, the optimization of scenarios and the tradeoff analysis thereof should take into account that not all scenarios are equally important to the user. For instance, a user might prefer to improve M1 for a given threshold, because it is very valuable for the business, while deferring the satisfaction of other (low-priority) scenarios.

3.3 Negotiation of Alternatives
In the approach explained above, each dbot executes its own local search and then shares its results with the other dbots (and with the architect). This schema is inspired by how a human architect (together with stakeholders) would do architecture exploration in real-life projects. Nonetheless, a drawback of SQuAT is that it might not provide all solutions and possible tradeoffs, as only a small subset of the Pareto frontier is returned by the dbots. If all candidate solutions explored by the dbots were to be provided (i.e., a larger subset of the design space), the user is likely to experience information overloading issues [18] that discourage her from making good decisions. In order to deal with this problem, we argue for a negotiation metaphor able to enlarge the current subset of alternatives with satisfying solutions. To reach a satisfying solution (in terms of quality-attribute tradeoffs), all the participants (humans, or dbots) should discuss and negotiate their architectural proposals.

In this context, we have developed a negotiation strategy for SQuAT, based on the monotonic concession protocol (MCP) for multiple parties [20]. This strategy generates "negotiated" solutions that complement the initial tradeoffs currently generated by the dbots. Basically, the outputs of the search are inputs to an MCP algorithm among the dbots. The dbots abide by a set of predefined rules that constitute the building blocks of MCP, namely: i) the agreement criterion, ii) which dbot makes the next concession, and iii) how much a dbot should concede. More formally, let $DB = \{db_1, db_2, ... , db_N\}$ be a set of N cooperative agents (dbots), and let $A_i$ be a finite set of architectural solutions (potential agreements) proposed by the agents. The set $A$ is interpreted as the union of the results of each $db_i$ when searching and applying tactics over the architecture. We denote $A_{i,j}$ to the ordered set of solutions for $arch_j$ computed by $db_i$.

The negotiation strategy depicted as pseudocode is provided in Figure 4. At first, each dbot makes an initial proposal with its first (best) solution. In our previous example, dbot M1 begins proposing solution A while dbot P1 begins proposing solution B. These initial solutions are interchanged to see if an agreement on any of the two solution can be reached. The notion of agreement is defined in terms of the utility of a proposal for the dbots. To this end, a utility

```
1: proposals = getInitialProposals(bots)
2: while not haveAgreement(proposals) do
3:    concedingBots = getBotsThatCanConcede(bots)
4:    if concedingBots.isEmpty() then
5:        informConflict()
6:        return
7:    else
8:        for concedingBots do
9:            newProposal = bot.makeConcession()
10:           updateProposal(proposals, newProposal, agent)
11:        end for
12:    end if
13: end while
14: informAgreement()
```

Figure 4: Negotiation pseudocode
function $U_i : A \rightarrow \mathbb{R}^+_0$ is assumed in each dbot $d_{bi} \in DB$ that maps agreements to non-negative values. The utility function $U_i$ gives a mapping between the scenario measures of an architecture $a_j$ from the point of view of $d_{bi}$. A utility function can be seen as the interest of a given stakeholder regarding a scenario. Note that the particular utility functions to be used in a given optimization must be specified by the architect.

There is an agreement if one dbot makes a proposal that is at least as good (regarding utility) for any other dbot as their own current proposals [10, 23]. If so, a proposal that satisfies all dbots is chosen. If not, a conflict exists. This criterion is a generalization of the agreement for bilateral negotiations [10]: an agreement is reached if there is a $d_{bj} \in DB$ such that $U_j(a_i) \geq U_j(a_j)$ for all $d_{bj} \in DB$, where $a_i$ is the last proposal of $d_{bi}$ and $a_j$ is the last proposal of $d_{bj}$. For instance, we can assume that $U(A) = 0.99$ and $U(B) = 0.98$, giving place to a conflict between proposals A and B. In case a negotiation round reports a conflict, one of the dbots must make a concession. A concession means that a dbot seeks a proposal with inferior utility in its list of solution. If no dbot can concede, the negotiation finishes with conflict, which means that no alternative tradeoffs are obtained from the negotiation. Selecting the dbot(s) that must concede is determined by the Zeuthen rule [23] (or willingness to risk conflict).

Various strategies are possible for the concession itself, i.e., the new agent proposal. For SQuAT we selected the egocentric concession, in which a dbot makes a proposal that is worse for itself (usually, the next solution in the dbot list). Since this kind of concession relies on the dbot’s own evaluation (rather than on that of other dbots), it does not force a dbot to know the utility functions of the other dbots. A dbot cannot concede anymore when it exhausted all the alternative solution in its list. In our example, after the initial conflict between A and B, let us assume that dbot M1 makes a concession and picks solution C from his list, which has a higher complexity but a lower utility for M1. Upon this proposal, dbot P1 agree on C and the negotiation succeeds.

4 EVALUATION

We conducted an experimental evaluation to investigate i) the kind of tradeoffs on PCM architectures produced by the search strategy, and ii) whether the negotiation strategy would lead to alternative (better) tradeoffs. While our approach is not limited to particular quality attributes, in this work we focused on modifiability and performance. The implementation of the SQuAT approach along with the models and data used for the evaluation are provided online². The remainder of this section covers the research methodology (Section 4.1), describes the scenarios and dbots under consideration (Section 4.2), and presents the main results and lessons learned (Section 4.3 and Section 4.4).

4.1 Methodology

The experimental procedure consisted of comparing the individual solutions of the dbots (using solely search) against the negotiated solutions (using search in tandem with negotiation). We configured one dbot per scenario (see Section 4.2). Along this line, we first ran the individual analysis (one per dbot) and plotted the local optima returned by the dbots. This search was configured with two levels of search, meaning that each dbot initially explored candidates for the initial ST+ architecture using only its tactics, then exchanged its candidates with those from the other dbots, and subsequently each dbot applied its tactics to those candidates (from other dbots). The first-level search produced 33 candidate solutions, while the second-level search produced 520 candidate solutions. This number of solutions is big enough to make the negotiation feasible. Second, we ran the negotiation strategy taking as inputs the $k$ solutions generated at each dbot and plotted the results.

For the negotiation, utility functions need to be configured in the the dbots. We defined a utility function (for all dbots) that takes into account the expected response (i.e., threshold) of the scenario $i$ ($ER_i$) and the scenario response for the alternative under analysis ($SR_i$). Our choice is a variant of the Boulware function [11], and we defined $U_i(a_j)$ as follows:

\[ (1) \text{ if } SR_i(a_j) \leq ER_i \text{ then } U_i(a_j) = 2 - \frac{ER_i}{SR_i(a_j)}; \]
\[ (2) \text{ if } SR_i(a_j) > ER_i \text{ then } U_i(a_j) = 2 - 1.10 \times \frac{SR_i(a_j)}{ER_i}. \]

When $U_i(a_j) > 1$ or $U_i(a_j) < 0$ the utility value will be 0. Thus, our utility function gives the maximum value to alternatives in which $SR_i = ER_i$ and penalizes alternatives in which $SR_i$ is far from $ER_i$. For example, if $ER=100$ and $SR=98$, then the utility is 0.98; however, if $ER=102$ the utility becomes 0.878. Note that while both $SR_i$s are at the same distance from $ER$, the utility of the first $SR_i$ (which satisfies the scenario) is higher than the other $SR_i$.

4.2 Scenarios and Dbots

The ST+ system was the initial architecture (Section 2). In total, we considered four scenarios, namely: two for modifiability ($M1$ and $M2$), and two for performance ($P1$ and $P2$), as presented in Table 1. The tactics for performance and modifiability configured in the respective dbots are listed in Table 2. Most of these transformations were described in [9], and in this work we adapted them to the PCM views. In function of the target quality attributes, our approach requires the selection of the analysis techniques to be implemented by the dbots as well as the configuration of predefined utility functions.

4.2.1 Modifiability. To assess modifiability, we relied on KAMP [22], which is a change impact analyzer for PCM. The analyzer takes a PCM repository model and returns all the components affected by a given change, either directly or by following its (rippling) effects to other components. In our case, a change corresponds to a modifiability scenario. As input, the architect has to provide a list of components (from ST+) being created, deleted, or modified in order to fulfill a new system feature. We extended KAMP to account for the complexity of components impacted by a change by adapting the cyclomatic complexity measure (of source code) [19] to work with SEFFs in PCM. Thus, each modifiability dbot can be configured to analyze a scenario by either computing the number of affected components or the component complexity. Regarding the optimization of repository models for modifiability, we could not find an existing solver for the task. Thus, we implemented PCM transformations using Henshin³ as modifiability

²https://github.com/SQuAT-Team/paper-supplementary/
³https://www.eclipse.org/henshin/
### 4.2.2 Performance

To assess and optimize for performance on PCM we relied on a combination of two approaches. First, for analyzing single PCM instances, we employed an existing model-to-model transformation resulting in a corresponding LQN model [16]. The LQN model is solved using the established LQNS tool [13] that produces predictions on performance measures such as response times and resource utilization. Second, we used PerOpteryx [14] to search for alternative candidates. Both parts were integrated in a performance dbot to be used in our SQuAT implementation. More technical details of the performance bot can be found in [12]. For the optimization, PerOpteryx employs a genetic algorithm to generate a set of alternative PCM instances based on an input PCM instance. In addition to the standard operations (selection, mutation, crossover), PerOpteryx includes performance-specific tactics to accelerate the search based on domain knowledge. The configured performance tactics are listed in Table 2.

### 4.3 Results

Figure 5 displays four charts with all the architecture candidates generated by the dbots. Each chart plots a pair of performance and modifiability scenarios. The X-axis represents the complexity of a candidate architecture (using our custom PCM measure displayed on a linear scale), while the Y-axis represents the response time (measured in ms and displayed on a logarithmic scale). The horizontal and vertical dashed lines are the thresholds of the performance and modifiability scenarios, respectively. Thus, within the shaded region both dbots satisfy their respective scenarios. The initial architecture for ST+ is drawn with a red point and labeled as "init". The green points (+) are candidates generated during the first-level search, while the blue points (×) are candidates generated during the second-level search. Since ST+ is a rather simple architecture and only contains a handful of components, both M1 and M2 dbots are satisfied from the start. However, the architectures do not satisfy dbots P1 and P2.

If each dbot would try to optimize the architecture in a selfish way, they could improve the initial results. For instance, the rounded boxes "m1", "m2", "p1" and "p2" are the best solutions for the scenarios (local optima), which were generated after applying tactics to the initial architecture. Unfortunately, performance-oriented changes actually hurt modifiability. For example, alternatives "p1" and "p2" help to move the response times below the scenario thresholds, but they also lead to not satisfying M1 because the addition of faster components to the architecture increases its complexity. Similarly, while alternatives "m1" and "m2" have positive effects on the satisfaction of M1 and M2 (which were already satisfied), they do not help for fulfilling P1 and/or P2. More interestingly, the optimization of one particular performance scenario does not mean that the other performance scenario gets improved. In fact, "p1" is a fast architecture for P1 but it achieves a poor response time with respect to P2.

With the negotiation strategy, the dbots start to make concessions looking for an agreement that mostly satisfies all the scenarios. The utility function is pivotal here, because it promotes alternatives being close to the scenario thresholds. In the charts, "a1" to "a10" stand for alternatives that were accepted by the four dbots after several negotiation rounds. Note that architecture candidate "a1", which is the first agreement obtained, is at the crossing of the thresholds of M1 and P1, M1 and P2, M2 and P1, and M2 and P2. This is a direct outcome of the utility function, because the candidate satisfies all the scenarios by a small margin, ensuring that the architecture is not over-optimized. This agreement achieved utility scores of 0.99, 0.97, 0.95, 0.75 for M1, M2, P1 and P2. The low score for P2 is attributed to the performance boost of "a1", which appears

### Table 2: Considered tactics and PCM models changed by the tactics

<table>
<thead>
<tr>
<th>Tactic</th>
<th>Repository</th>
<th>Assembly</th>
<th>Infrastructure</th>
<th>Allocation</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Modifiability</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Split component: Divide interface and component, e.g., in case of low cohesion.</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>(Y)</td>
</tr>
<tr>
<td>Add wrapper/intermediary: Wrap a component by a façade, e.g., to add functionality.</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>(Y)</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Component selection: Selection of alternative components that provide and require the same interface(s).</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Component re-deployment: Re-deploy in case load is not evenly distributed or many remote calls are detected.</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Resource scaling: Change processing rate in case high resource utilization is detected.</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Moving distributed components: Move components to a single server in case many remote calls are detected</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Change passive resources: Capacity of passive resources are changed.</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Selection, mutation, crossover: Default genetic operations based on automatically determined degrees of freedom.</td>
<td>(Y)</td>
<td>(Y)</td>
<td>(Y)</td>
<td>(Y)</td>
<td>N</td>
</tr>
</tbody>
</table>
in the chart as a small distance from the threshold that is penalized by the function.

In terms of optimization, candidate “a1” is initially improved for performance by dbot P2, making changes to the repository, system assembly, allocation, and resource environment models. First, the components QuickBooking and AlternativeExporter are added to the repository. Then, runtime elements in the system assembly model are modified to employ those alternatives in runtime. In the allocation model, assembly contexts for BusinessTripMgmt and AlternativeExporter are moved from Server1 to Server2 and DB Cluster, respectively. At last, the deployment model is tuned to cope with scenario P2, by modifying the processing rates of the nodes. Basically, the HDD processing rate of Server 1 is reduced due to the migration of AlternativeExporter to another server, and the CPU processing rates of each of the two nodes in DB Cluster are increased from 6 to ≈11, to tolerate failures such as the unavailability of one DB node and support the newly-allocated component. Regarding modifiability, dbot M1 splits component Payment System into two subcomponents (one for interface IExternalPayment and another for IEmployeePayment) and wraps the interface IExternalPayment, reducing the ripple effects of changes.

In case “a1” is not an appealing candidate to the architect, she can resume the negotiation in order to further explore more agreements (up the point that the negotiation ends with a conflict). These additional agreements are marked by the remaining blue labels in the charts. An interesting alternative is “a5”, which is the result of several optimizations. Differently to “a1”, in “a5” the wrapped component is ITripDB. In the system model, Booking, TripDB and
A number of approaches have tackled the optimization of software architectures in function of quality attributes. A survey and a general optimization process are presented in [3]. Most approaches are limited to a few quality attributes with a monolithic optimization design, i.e., the addition of new quality-attribute knowledge is not straightforward. Representative examples of architecture optimization platforms that include some flexible or distributed design aspects are AQOSA [17], ArchE [4, 9], ArcheOpteryx [2], DesignBots [8], and PerOpteryx [15].

ArchE [4, 9] is an assistant for architecture design that inspired several design concepts used by SQuAT. ArchE proposes the notion of reasoning framework as a modular unit that encapsulates the knowledge about a particular quality attribute. ArchE searches through the design space via a centralized rule-based engine that relies on services of the available reasoning frameworks, in order to direct the search towards solutions that satisfy quality-attribute scenarios. Internally, each reasoning framework implements a set of tactics. The analysis of quality-attribute tradeoffs is similar to that of SQuAT, but ArchE does not include negotiation; thus, only a limited number of tradeoffs based on local optima can be returned to the architect.

DesignBots [8] introduce the notion of agents as wrappers for the ArchE reasoning frameworks. These agents have more autonomy than the reasoning frameworks in order to fulfill their own goals. In DesignBots, the design exploration problem is viewed as a mixed-initiative planning model, in which scenarios are goals attainable by planning operators that capture architectural tactics. Along this line, each agent returns a plan to modify the current architecture, and there is a coordination agent responsible for merging the individual plans into a global plan, which is presented to the architecture. The tradeoffs are the results of this global plan.

AQOSA [17] provides tools for component-based systems that allow for design space exploration. Scenario-based analyses for performance, reliability and cost serve to focus the design on particular architectural configurations. The developer can visualize the resulting architectures using Pareto curves, which are used for making design tradeoffs explicit to the architect. A drawback of DeSiX is that it does not support automated search, and the developer manually selects possible configurations to be evaluated by the tool.

ArcheOpteryx [2] is an optimization framework targeted to the Architecture Analysis & Design Language (AADL) for embedded systems. The optimization engine is based on genetic algorithms and is amenable to perform multi-objective optimization. The optimization of each particular objective (or quality attribute) is handled by a separate AADL analyzer. The optimization is not based on tactics but rather on traditional mutation and crossover operators of genetic algorithms. The optimization results can be visualized as Pareto diagrams, although the architect cannot steer the search for particular quality-attribute tradeoffs, as our negotiation approach aims to do.

PerOpteryx [15] is an optimizer for architectures modeled with PCM. PerOpteryx is equipped with sophisticated quality-attribute analyzers for performance, reliability, and cost. PerOpteryx is also based on genetic algorithms, but it extends the mutation and crossover operators with an additional operator for capturing architectural tactics. In this way, it is possible to incorporate tactics into the optimization process. Nonetheless, the original conception of PerOpteryx is monolithic in nature, and cannot take advantage of modular or distributed approaches, such as those of SQuAT.

Another related approach for tradeoff analysis in engineering domains is MATE (Multi-Attribute Tradeoff Exploration) [21]. This approach proposes a system design process based on notions of need identification, architecture exploration, and evaluation, which have direct correspondences with the notions of scenarios, tactics and analysis models of SQuAT. MATE does not have utility functions for assessing the alternatives being considered. However, we believe
that the SQuAT framework can be improved with some tradeoffs concepts from MATE.

6 CONCLUSIONS AND OUTLOOK

Satisfying quality attributes is a key aspect during software architecture design. In this paper, we presented the SQuAT approach that assists architects in exploring design alternatives focusing on their tradeoffs. The exploration was realized by means of a heuristic search strategy followed by a negotiation strategy, which is a novel aspect of SQuAT for handling tradeoffs. Specifically, we exercised our framework with an analysis of modifiability versus performance in PCM architectures. Our preliminary results showed that the negotiation strategy can provide an interesting, but still manageable, set of tradeoffs to the architect.

However, our approach still has some limitations that are worth discussing. A first concern is the validity of the parameters in the quality-attribute models and utility functions used in the experiments. Both the performance and modifiability analyzers are designed to provide a variety of results, but these results are based on assumptions that may be hard to achieve. For instance, for performance, resource demands have to be specified as part of the components’ SEF in the PCM repository. In the design phase, these values can only be estimated by experts but can be refined and validated based on dynamic analysis of implementation artifacts in later development stages [1]. Similarly, the utility functions need to be validated against architects’ preferences or stakeholders’ interests. Furthermore, other implementations (i.e., a different analysis technique) are possible for the modifiability dbot or the performance dbot used in our case study. The focus of our evaluation was to demonstrate the feasibility of the concepts, but it was not intended to assess the validity of the analyses for the scenarios or the structural quality of the design alternatives presented to the architect. We acknowledge that the solutions automatically generated by the dbots, even when maximizing the quality-attribute measures, might not be readily accepted by human architects, because these solutions might not match their preferences.

We plan to address this threat by conducting larger case studies, ideally using real-world systems and scenarios. Apart from the utility functions, the negotiation might rely on priorities of the scenarios, different concession strategies, or require additional search for particular solutions [20]. Another idea is to define user profiles with stakeholders’ preferences and link those profiles to the dbots, so as to incorporate those preferences in the search and negotiation algorithms. Second, SQuAT requires a complete definition of the architecture using PCM, the scenarios (e.g., their response measures and mappings), and the architectural tactics to be considered. These artifacts might exist just partially in practice. In some cases, like performance, architectural models can be extracted automatically from performance measurements [1], e.g., obtained in test or production environments. The combination with automatic extraction techniques makes SQuAT a candidate to support optimizations at runtime as part of self-adaptive software systems.

Apart from the previously outlined directions, we plan the following future works: i) incorporate other quality attributes, such as reliability and security, ii) improve the scenario measures and their validation, iii) experimental comparison with non-distributed approaches, iv) removing the current dependency to PCM, and v) including human feedback during the optimization.

ACKNOWLEDGMENTS

This work is being supported by the Argentinian Ministry of Science, Technology and Productive Innovation (MINCYT), the German Federal Ministry of Education and Research (BMBF, grant no. 01DN15014), and the German Research Foundation in the Priority Programme “DFG-SP 1593: Design For Future—Managed Software Evolution” (DFG, grant no. HO 5721/1-1).

REFERENCES


