Can Multi-Agent Consensus Improve Quality Tradeoffs in Software Architecture Optimization?

J. Andres Diaz-Pace[1], Santiago A. Vidal[1], Antonela Tommasel[1], Sebastian Frank[2], Andre van Hoorn[2]

[1]ISISTAN Research Institute – UNCPBA University & CONICET
Tandil – Buenos Aires – Argentina

[2]University of Hamburg – Hamburg – Germany

{andres.diazpace,santiago.vidal, antonela.tommasel}@isistan.unicen.edu.ar
{sebastian.frank,andre.van.hoorn}@uni-hamburg.de

Abstract. *When designing a software architecture to fulfill quality-attribute requirements, architects normally explore and assess alternative solutions leading to different quality-attribute tradeoffs. In this context, we developed the* `SQuAT` *framework to support automated multi-objective optimization in large design spaces.* `SQuAT` *provides a modular, multi-agent architecture in which each agent represents and optimizes a particular quality attribute. However, this search strategy has problems identifying tradeoffs that satisfy all the parties (or architects' concerns), particularly when searching for many candidate solutions and evaluating them becomes computationally costly. This is actually a general challenge for architecture optimization tools. To deal with it,* `SQuAT` *features an agent negotiation protocol that seeks consensus based on the utility of solutions as judged by each agent. In this paper, we present a parameterized heuristic that enhances the integration between search and negotiation in* `SQuAT`*, and also report on an empirical evaluation with two case studies. The results show initial evidence that using negotiation is more effective than doing a pure search to identify solutions having a balanced utility across agents, and thus, offer alternative quality-attribute tradeoffs to the architect.*

1. Introduction

The software architecture of a system is a blueprint that captures the main design decisions for satisfying the functional requirements and quality attributes (e.g., performance, modifiability, reliability, among others) posed by the system stakeholders [Bass et al. 2021]. Designing a software architecture in such a way it meets its quality-attribute requirements can be a complex process, as architects often need to consider alternative solutions satisfying the same requirements, and furthermore, these solutions lead to tradeoffs. A tradeoff means that a given architecture (or any of its design decisions) has a positive impact on a given quality attribute at the cost of negatively affecting another one — e.g., separating functions into modules favors modifiability but might have a performance impact.

Given an initial architecture, the number of possible solutions for improving it (with respect to predefined quality-attribute requirements) can be too big to be assessed by a human, even for medium-sized problems. For this reason, automated optimization

approaches for the architecture domain have been developed [Aleti et al. 2013]. The optimization process is normally cast to a heuristic search driven by multiple objectives and constraints, in which an initial architecture undergoes different transformations (or refactorings). As output, this search is expected to produce a number of architecture candidates with different tradeoffs, according to the architect's criteria as the decision maker. An instance of such optimization approaches is the SQuAT framework [Rago et al. 2017], which relies on a multi-agent architecture organized into separate modules (agents) called *dbots*. SQuAT takes a *model-based approach* and uses the Palladio Component Model (PCM) [Becker et al. 2009] for representing software architectures. Departing from an initial PCM architecture, each *dbot* performs a local search, evaluating and optimizing candidates with respect to a single quality attribute. Afterward, the *dbots* rely on a negotiation protocol to identify candidates that globally satisfy all the qualities. To do so, the *dbots* try to reach "agreements" (or consensus) [Rekha and Muccini 2018] based on the utility of the solutions explored by each agent.

The SQuAT framework has evolved over the years, and encouraging experiments with case studies involving modifiability and performance have been reported. However, applying the approach in larger, more real architectural settings brings several challenges. A first problem refers to the *computational cost* of the optimization. For example, analyzing a PCM architecture with a performance analytical solver [Koziolek and Reussner 2008] to determine whether an objective is met can take a considerable amount of time (from seconds to minutes), and this effort increases as more objectives and architectural candidates are evaluated during the search. This is the case when multiple architectural transformations are explored, increasing the depth of the search. Therefore, pruning must be considered to avoid a combinatorial explosion of candidates while still being able to return a set of "interesting" solutions to the architect. A second problem is about the utility of solutions regarding their quality-attribute tradeoffs. In practice, architects might not be interested in optimal solutions (in the sense of maximizing or minimizing objectives or reaching the Pareto front) but in *satisfying solutions*, i.e., solutions being good enough for the objectives (or other architects' preferences) tackled by the *dbots*. Within SQuAT, this means searching for candidates having *balanced* tradeoffs, that is, architectures providing more or less the same utility to all the agents. We should note that these challenges are not unique to SQuAT and apply to any architecture optimization tool [Di Pompeo and Tucci 2023].

A (pure) search strategy cannot always produce balanced solutions unless a significant portion of the design space is explored. However, as mentioned above, increasing the computational resources (for the search) is not always feasible. This is where the negotiation strategy comes into play. We argue that a judicious combination of search and negotiation can help steer the search toward solutions with balanced tradeoffs without a large exploration of the design space. This idea was proposed in previous work [Monteserin et al. 2017], but the combined strategy has neither been fully developed nor empirically evaluated yet. Here, we formalize and enhance the strategy as a heuristic that depends on several parameters. We also analyze its behavior by simulating search traces in two PCM design spaces: *ST+* and *CoCoME*. Although preliminary, the results show that the *dbots* using negotiation are able to find many solutions with balanced tradeoffs while keeping the number of examined solutions bounded, which is an advantage over pure search. An adequate strategy parameterization and the complexity of

the optimization space are also drivers to consider. Beyond SQuAT, we believe that the techniques and insights from this work contribute to improving the usage and adaptation of optimization techniques in the software architecture domain.

The rest of the paper is organized as follows. Section 2 provides the basic concepts of architecture optimization using *ST+* as a motivating example and briefly introduces SQuAT. It also covers related work. Section 3 presents the search heuristic integrated with negotiation and its main parameters, which is the central part of this work. Section 4 reports on the evaluation of the combined strategy for the *ST+* and *CoCoME* case studies. Finally, Section 5 gives the conclusions and outlines future work.

2. Background

To contextualize the SQuAT concepts, we describe a quality-driven optimization problem for software architectures and how it can be tackled using search techniques and architectural transformations. We also introduce a simplified version of the *ST+* system [Koziolek 2013, Rago et al. 2017] to illustrate the concepts.

2.1. Motivating Example

Extended Simple Tactics (*ST+*) [Rago et al. 2017] is a trip management system, adapted from [Koziolek 2013], in which users can search and book trips with different payment options and request reimbursements for cancellations. The system architecture is specified in PCM, as shown in Figure 1. In PCM, the architecture consists of several models, each representing an architectural view: *i)* the component types and interfaces (repository), *ii)* the system assembly, and *iii)* the deployment of this assembly to the infrastructure (e.g., servers interconnected via network links). *ST+* also includes a behavioral specification for the component implementations in the repository and a usage model with an open workload for the system-provided services. For brevity, these two models are not shown.

In the repository (top level), ST+ has several components and interfaces available. BUSINESSTRIPMGMT is the system entry point for the users. BOOKINGSYSTEM is invoked for handling a booking. PAYMENTSYSTEM is called to process payments or reimbursements. QUICKBOOKING is an alternative to BOOKINGSYSTEM. ST+ saves trip (and reimbursement) information 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 EXPORTER. In addition, since logos must be added to a PDF file, ALTERNATIVEEXPORTER stores them in a compressed version. For the initial architecture (middle level), a subset of the above components is instantiated and assembled: BUSINESSTRIPMGMT, BOOKINGSYSTEM, PAYMENT SYSTEM, EXPORTER, and TRIPDB. The deployment infrastructure (bottom level) comprises three servers (SERVER1, SERVER2, and SERVER3) and a DB CLUSTER with two nodes connected via a LAN. The allocation of components to servers is depicted with black arrows.

In the example, we assume that *ST+* must fulfill stakeholders' performance and modifiability requirements. These requirements are captured by *scenarios* [Bass et al. 2021], which specify objectives in quantitative measures (e.g., change effort for modifiability or response time for performance) that architectures must meet. Let us consider two scenarios (i.e., optimization objectives) for *ST+* as follows:

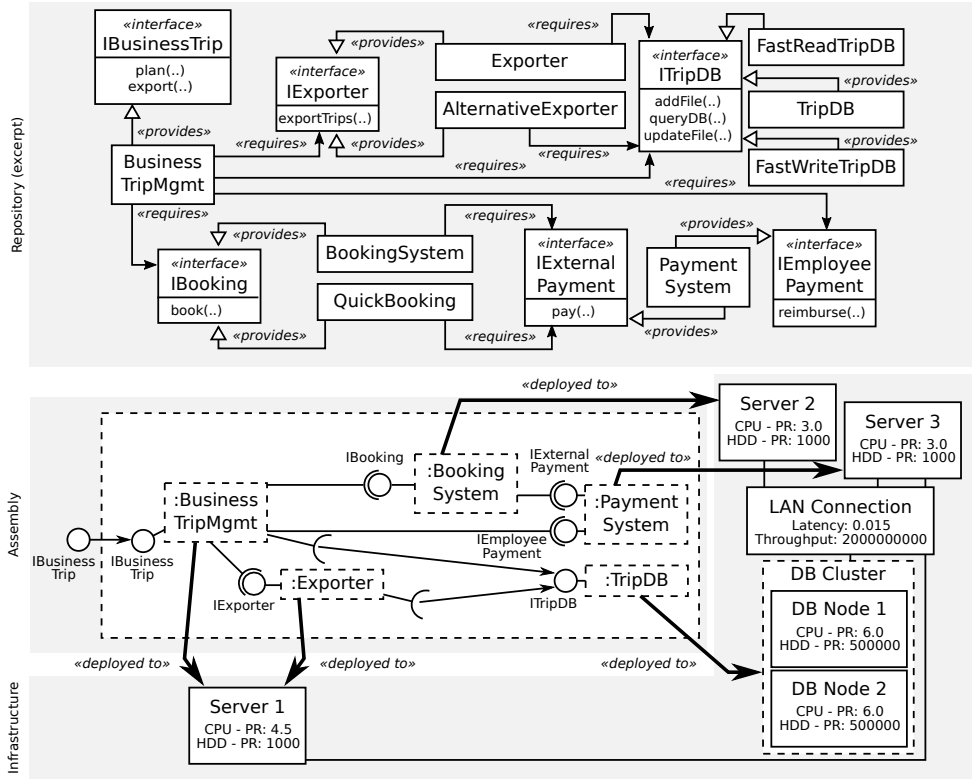


Figure 1. PCM models for the *ST+* system (initial architecture) [Rago et al. 2017].

• **M1** (modifiability) refers to adding a new payment option and specifies that the number of affected components should be less than 5, with a total complexity of 120.

• **P1** (performance) is about the system workload and requires the response time to stay below 30 ms.

In the scenarios, their quality-attribute measures are treated as satisfaction thresholds; that is, any architecture whose performance and modifiability scores are below 120 and 30, respectively, are considered *satisfying* solutions for the problem. We refer to the space of quality-attribute measures as the *quality-attribute space*. To determine the quality-attribute scores of PCM architectures, we rely on existing analysis models, such as (layered) queueing networks for performance [Koziolek and Reussner 2008] and change impact analysis for modifiability [Stammel and Reussner 2009]. In this way, we can assess how far the *ST+* initial architecture (A_0) is from satisfying the thresholds in the quality space. Let us assume that quantitative analyses are run on A_0, and it turns out that A_0 satisfies $M1$ but does not satisfy $P1$, as shown in Figure 3. Thus, the architect would like to explore candidate solutions for improving both scenarios, as denoted by the blue area in the figure, which we refer to as the *satisfaction area*.

2.2. Architectural Transformations and Search Tree

One way of generating candidate architectures is through design transformations, also known as *tactics* [Bass et al. 2021]. In this paper, we are interested in transformations targeting modifiability and performance objectives. To improve modifiability, architects have several options at their disposal. For instance, the transformations 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

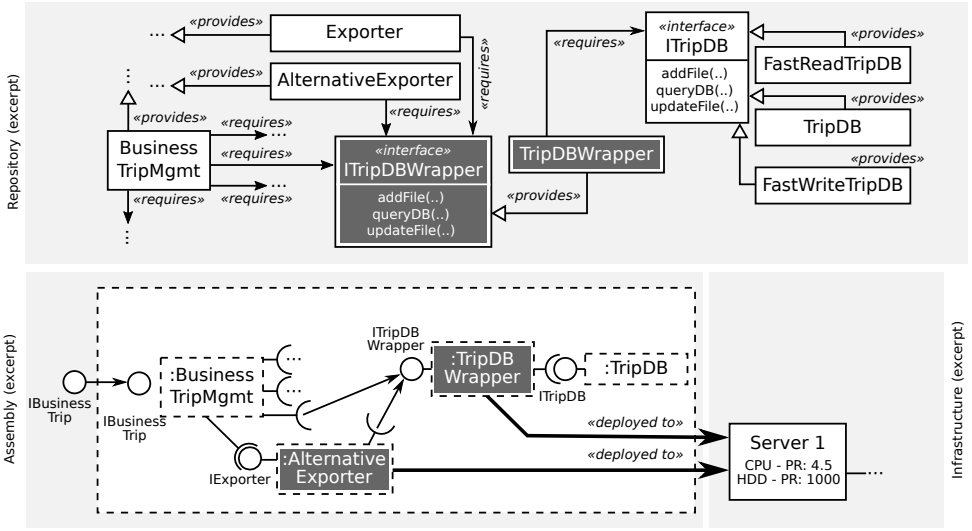


Figure 2. Modifiability and performance tactics applied to *ST+* [Rago et al. 2017].

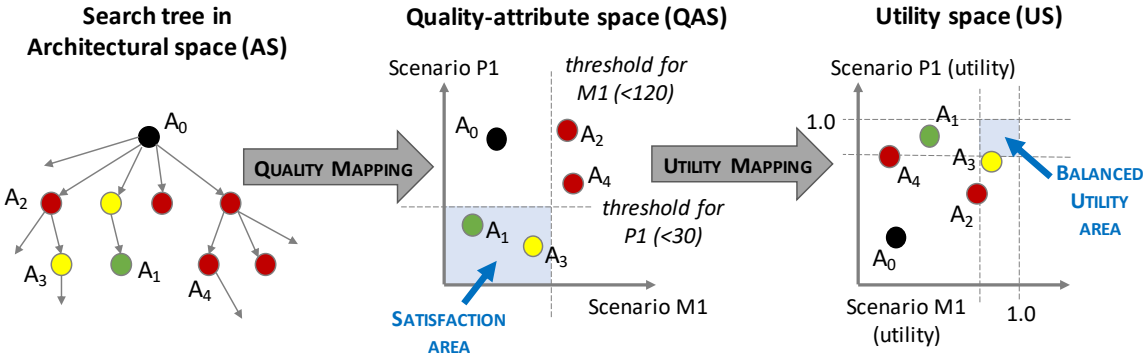


Figure 3. Relation between search tree (left-side), quality-attribute space (center) and the utility space (right side). The green circles refer to architectures that satisfy all the quality-attribute scenarios, while the red circles correspond to architectures that do not satisfy any of the scenarios. The yellow circles indicate that the architectures satisfy some of the scenarios.

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 2 (top). A wrapper component and an interface (ITRIPDBWRAPPER and TRIPDBWRAPPER) are created to avoid rippling effects after changing ITRIPDB operations. This tactic might cause a performance drop since another component is deployed to the servers, and the component communications with the DB are indirect through the wrapper. The candidate architecture resulting from the wrapper tactic is denoted as A_1 in Figure 3.

From a performance perspective, possible tactics for the architect are: i) select a particular component alternative, ii) allocate components to different servers, or iii) introduce performance-specific components such as a cache or a load balancer, among other transformations. Figure 2 (bottom) shows the replacement of the EXPORTER component by the more efficient ALTERNATIVEEXPORTER component, which ensures that the additional load (as required by *P1*) can be handled. Applying this performance tactic impacts modifiability, e.g., due to additional maintenance efforts. The candidate architecture resulting from the replacement tactic is denoted by A_2 in Figure 3.

The successive application of tactics (from the initial architecture) gives place to

a search tree, as sketched on the left of Figure 3. This tree is constructed by an optimization engine (actually, by the collaborative work of the *dbots*) in order to explore candidate solutions. The nodes correspond to candidates, while the directed edges capture the transformations from a candidate to its children in the search. A tactic normally comes with preconditions that the architecture on which it is applied must fulfill. Thus, not all tactics can be applied to all architectural candidates. Those candidates that satisfy any or all the objectives (scenarios) are architectural solutions to be presented to the architect. These solutions will have different tradeoffs with respect to the objectives. In principle, the tree can grow in breadth or depth, and become quite large, even for small design spaces. Controlling how the search tree gets expanded is one of the challenges when designing heuristics for architecture optimization [Quesada et al. 2018, Di Pompeo and Tucci 2023].

Although getting candidate solutions that fall in the satisfaction area (center of Figure 3) is one of the optimization drivers, not all solutions might be useful to the architect. To this end, we introduce the notion of *utility* to capture architects' preferences and also avoid situations of "over-optimization", given the computational costs of the search. Thus, a utility function converts the quality-attribute space into a *utility space*, in which 1 indicates maximal utility and 0 indicates no utility. Figure 3 illustrates how A_0, A_1, and A_2 get mapped to different utility values. For instance, we can have a utility function (for both $M1$ and $P1$) that tends to 1 as the quality-attribute values are within a short distance of the scenario thresholds and decreases towards 0 otherwise. Along this line, the candidate solutions are quantified in both the quality-attribute and utility spaces. Assuming that all scenarios must be (mostly) satisfied, an ideal solution should get a utility in the right-upper quadrant of the utility space. Furthermore, such a solution should have a *balanced* utility across all scenarios, meaning that it is useful to all parties. This utility space region is regarded as the *balanced utility area*, as marked in Figure 3. Because of the discrete nature of the (feasible) architectural transformations, finding these candidates is not straightforward. Not all satisfying candidates will necessarily have a balanced utility. In practice, the search process might eventually find candidates with a balanced utility, but a challenge is how to increase the chances of finding them.

2.3. Quality-driven Architecture Optimization

We consider a multi-objective architecture optimization along the lines of the PAD (Predictable Architecture Design) model [Bachmann et al. 2005, Pace and Campo 2008]. We assume an *architectural space* for a family of systems that encompasses all possible solutions in terms of a (finite) set of *design decisions*. More formally, let $AS = \{A_0, A_1, A_2, ..., A_n\}$ be a space in which each A_i corresponds to a (valid) architecture resulting from predefined design decisions $A_i = \{d_{1i}, d_{2i}, ..., d_{mi}\}$. In this work, we restrict the possible decisions to *architectural tactics* [Bass et al. 2021]. A tactic is a transformation that affects parts of an architectural structure (e.g., PCM components and interfaces) to improve a particular quality attribute. Thus, the architectures in AS are linked to one another through the application of tactics. AS can be visualized as a directed graph in which each node represents an architecture, while an edge between two nodes A_i and A_j captures a tactic from the former to the latter. A tactic T_ij changes the architectural structure of A_i to become that of the target A_j.

An architecture A_i can be assessed with respect to multiple quality attributes (objectives) through quantitative measures (or scores). Let $QAS =< O_1, O_2, ..., O_m >$ be

a *quality-attribute space* in which each objective O_k represents a quality-attribute metric (e.g., response time, change complexity, etc.) associated with some candidate in AS. Hence, the evaluation function $f : AS \rightarrow QAS$ maps architectures to multi-valued vectors in $\Re^m$. Furthermore, let $IS =< I_1, I_2, ..., I_p >$ be a *utility space* in which I_k represents a utility metric in the range $[0..1]$ for some multi-valued vector in QAS. Consequently, we have a utility function $u : QAS \rightarrow IS$ that maps quality-attribute values to utilities. Note that $u = f(AS)$. In Figure 3, the quality-attribute values and utilities for the architecture examples are 2-dimensional vectors in QAS and IS, respectively. In the general case, all these spaces can be n-dimensional.

2.4. Related Work

Two approaches related to our work are *EASIER* [Arcelli et al. 2018] and *GATSE* [Procter and Wrage 2019], which implement a multi-objective search strategy using evolutionary algorithms. The rationale of this technique differs from that of the *dbots*, in the sense that a collection of multiple candidates (population) is evolved through several iterations (generations). Furthermore, the architecture solutions are encoded as chromosomes and transformed by means of genetic operators. On the one hand, *EASIER* relies on architectural specifications written in *AEmilia*, which enables performance and reliability analyses. They also consider the number of architectural changes from one candidate to another as an additional objective to be minimized. On the other hand, *GATSE* provides a visual explorer for *AADL* (Architecture Analysis and Design Language) specifications that displays quality-attribute analyses of selected architectural configurations, enabling the architect to focus on regions of the quality space and progressively narrow down the search, also referred to as "design by shopping". Like in PCM, the usage of model-based analysis techniques for quality attributes is a shared characteristic of the specifications used by *GATSE* and *EASIER*, respectively.

Unlike the previous approaches, our work focuses on quality-attribute tradeoffs and tries to manage them (in the sense of reaching a consensus among parties), instead of seeing tradeoffs from the traditional view of a Pareto frontier. The usage of negotiation is a novel aspect, which has the potential of exploring different tradeoffs than an evolutionary search. Furthermore, the improvements for `SQuAT` discussed in this paper can reduce the computational costs of the optimization of architectural models, which is currently a challenge for *GATSE* and *EASIER*, among other optimization approaches.

3. Integrating Search and Negotiation

In general, an automated heuristic is needed to drive the exploration of the design space and generate alternative solutions according to the architecture optimization formulation. `SQuAT` provides a multi-agent modeling approach and a modular architecture for PAD, which supports various search strategies. The main aspects of the framework for achieving this goal are explained next — the reader is referred to our previous work [Rago et al. 2017] for more details. This work focuses on mechanisms to address the two challenges discussed in previous sections. Specifically, we expect to obtain architecture candidates at the intersection between the satisfying and balanced utility areas while reducing the computational overhead of the (pure) search.

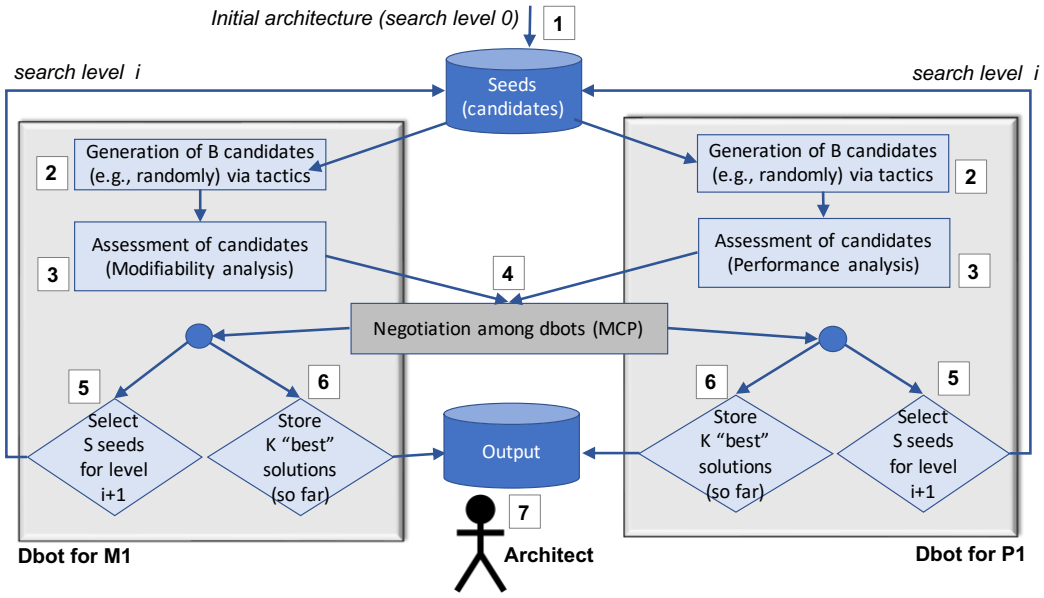


Figure 4. Distributed search process (individual *dbots*) and negotiation phase.

3.1. *SQuAT* Overview

The `SQuAT`[1] framework is structured around the notion of *dbots* — agents encapsulating the required knowledge to analyze and improve a particular quality attribute [Rago et al. 2017]. This knowledge involves: *i)* an automated analyzer (or solver) that takes an architectural description (e.g., PCM) and computes a quality metric, and *ii)* a set of tactics implemented as architectural transformations. Unlike other architecture optimization tools, the rationale behind SQuAT is that each *dbot* performs a single-objective optimization with little knowledge about the objectives being pursued by other *dbots*. From a modeling perspective, a *dbot* is intended to capture the interests of a particular stakeholder regarding the architecture. These interests are expressed by means of quality-attribute scenarios [Bass et al. 2021] and utility functions.

The search strategy followed by the *dbots* is summarized in Figure 4. For simplicity, we assume two agents optimizing scenarios $M1$ and $P2$, respectively, although it can be generalized to n agents. Each agent accesses a storage, called `Seeds`, with the candidates to be explored and potentially improved in its next round of search (or tree level). There is another storage, called `Output`, in which all agents persist their best candidates so far. These are the candidates presented to the architect once the process ends. A round of search consists of seven steps, namely:

1 An initial architecture specified in PCM (and provided by the architect) is configured as the first seed (candidate) to trigger the *dbots* activity. In intermediate rounds of search, the *Seeds* storage is updated with candidates from the *dbots*, as denoted by the feedback loops in the figure.

2 Each *dbot* takes the first architecture from the *Seeds* storage (using some priority criterion) and generates several children candidates by applying predefined tactics to different parts of the architecture. The *dbot* $M1$ will only use modifiability tactics, while the *dbot* $P1$ will only use performance ones. Within a *dbot*, this step entails a local, single-objective search. The *dbots* also check that the resulting configuration of the candidates is valid; otherwise, the child is discarded.

[1] *SQuAT* is an acronym for *S*earch Techniques for Managing *Qu*ality-*A*ttribute *T*radeoffs in Software Design Optimizations. It started in 2016 as part of an international collaboration project.

3 The *dbot* runs a quality-attribute analyzer to get quantitative values for both the seed architecture and its children. Analogously to the previous step, the *dbot* $M1$ will rely on a modifiability analyzer (e.g., [Stammel and Reussner 2009]) and the *dbot* $P1$ will rely on a performance analyzer (e.g., [Koziolek and Reussner 2008]). Executing these analyzers usually involves a computational overhead. After the analysis, each *dbot* ranks its candidates according to quality-attribute (or utility) criteria.

4 The *dbots* engage in a negotiation protocol based on the available candidates, their scenario thresholds, and utility function. This step simulates a human negotiation about (the utility of) the candidates, in which some agents might be willing to accept inferior candidates (in their rankings) to reach an agreement with the other agents.

5 Each *dbot* gets a new ranking for its candidates, which might include candidates agreed during the negotiation. At this point, a set of candidates must be selected as seeds for the next round of search. These candidates are chosen using the same criteria from step 3. The process is repeated from step 2.

6 In parallel, each *dbot* gets a new ranking for its candidates. At this point, a set of "interesting" candidates must be sent to the `Output` storage as intermediate solutions of the current round. These candidates are expected to have quality-attribute tradeoffs and (ideally) balanced utility.

7 Once the maximum number of tree levels is reached, the process is finished, and all the stored solutions are presented to the architect.

3.2. The Negotiation Protocol

In step 4, the *dbots* use the Monotonic Concession Protocol (MCP) for negotiation [Endriss 2006]. We give here an intuition of how MCP works since full coverage is out of scope — the MCP instantiation for `SQuAT` can be found at [Monteserin et al. 2017]. Essentially, MCP supports agent coordination allowing the *dbots* to exchange and retain candidates (either as seeds or outputs) that could be otherwise disregarded. The goal is to keep solutions with better (or more useful) tradeoffs than those obtained through the agents' local search.

In MCP, the *dbots* are assumed to be cooperative and abide by rules about: *i)* the agreement criterion; *ii)* if no agreement is achieved, which *dbot* makes the next concession; and *iii)* how much a *dbot* should concede. Initially, each *dbot* makes an initial proposal with its first (best) candidate for the current search level. All proposals are interchanged to check if an agreement on any of them can be reached. The agreement is defined in terms of the proposal utility for the *dbots*. As mentioned previously, a utility function $u_i(A_j)$ that assigns non-negative values to the proposals is assumed. There is an agreement if one *dbot* proposes a candidate A_x that is at least as good (regarding utility) for any other *dbot* as their own current proposals [Endriss 2006]. If so, A_x is chosen, and the negotiation ends successfully. If not, a conflict exists, and some *dbot* should make a concession. A concession means that a *dbot* seeks an alternative proposal in its list of candidates. If no *dbot* can concede, the process finishes with conflict, which means that nothing is obtained from the negotiation. The selection of the *dbot* that must concede is determined by the Zeuthen rule [Endriss 2006], which evaluates each dbot's willingness to risk conflict and prefers a conflict-free proposal being close to Nash equilibrium. As for the concession itself, a *dbot* makes a new proposal that is worse for itself (usually, the next candidate in its ranking).

For example, coming back to Figure 3, let us assume that in level 2 the *dbots* $M1$ and $P1$ enter a negotiation with A_1 and A_3 as proposals from $M1$ and A_4 from $P1$. A_1 seems like a good candidate in terms of quality, but let us suppose that its utility is not good for $P1$, and the *dbot* proposes A_4 instead. Although this proposal is closer to the quality thresholds, let us suppose it conflicts with the expected utility by $M1$. At this point, $M1$ makes a concession and picks A_3

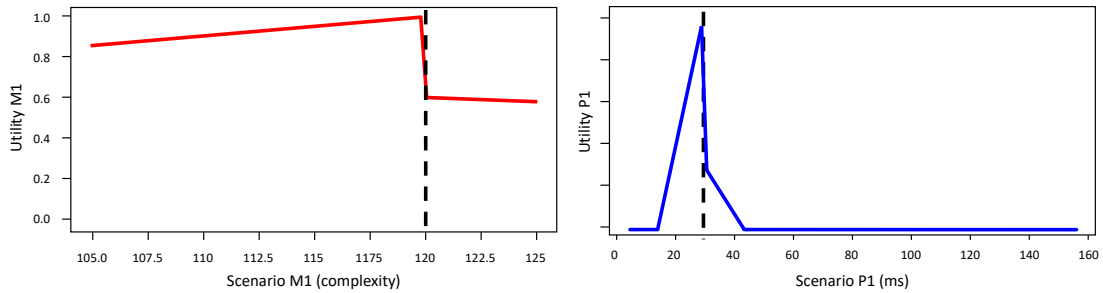


Figure 5. Examples of utility functions (the dashed lines are the thresholds).

from its ranking. Upon this candidate, which has a higher complexity but a slightly lower utility for $M1$, $P1$ and $M1$ reach a utility consensus, and the negotiation succeeds.

For the negotiation strategy to work, utility functions need to be configured in the *dbots*. We defined a utility function (for all *dbots*) that takes into account the desired scenario threshold and the actual scenario value for the candidate under analysis. This is a variant of the Boulware function [Fatima et al. 2004], which gives the maximum value to candidates whose quality value is equal to the threshold, as exemplified in Figure 5, and penalizes values otherwise (with different slopes). Other parameterizations of the function or alternative utility functions are possible.

3.3. Proposed Strategy

To deal with projects of different complexity, we designed the search process of Figure 4 as a parameterized heuristic. The parameters can be seen as knobs to control the expansion of the search tree, the tactics being considered at each level, and the criteria to rank certain candidates while discarding others. The heuristic is shown by Algorithm 1 and the parameters are below.

- *Branching factor (`B`):* It gives the maximum number of children a *dbot* is allowed to create for a given seed, with a predetermined set of available tactics (step 3). Candidate generation via tactics is performed randomly until the B limit is reached (line 8 in the algorithm). Large B values can promote candidate diversity at the cost of increasing the number of candidates being evaluated.

- *Ranking criteria:* It is the *dbot*'s rule to sort candidates (at a given level). The default criterion, which takes a quality-attribute perspective, is to rank first those candidates with the smallest quality value (assuming objective minimization). An alternative criterion is to look at the utility of the quality values, ranking candidates with the highest utility first.

- *Seeding factor (`S`):* It determines how many children candidates, once generated and assessed, a *dbot* can pass on as seeds for the next level (step 5). Some of these candidates might come from the negotiation phase. S controls the tree growth in subsequent levels of search (line 12 in algorithm). Each *dbot* retains the top-S candidates and deletes the rest, using the `filteringCriterion`. It can be configured either as quality- based or utility-based criterion.

- *Retaining factor (`R`):* It limits the number of candidates a *dbot* can return at a given level of search. The union of all the candidates (by all the *dbots*) is presented to the architect at the end of the process (step 6). This parameter might be useful to filter the *dbots*' output and reduce information overload on the architect. The `outputCriterion` is employed to filter the solutions (line 13 in algorithm), which can also be quality-based or utility-based.

- *Tree depth (`maxLevel`):* It bounds the search tree to a maximum number of levels.

Determining the right parameters for the heuristic is not straightforward, because it depends on project-specific characteristics (e.g., a family of architectures and scenarios).

Algorithm 1 Heuristic search for architecture candidates.

Inputs: A_0 : initial architecture
$dbots$: available *dbots* (one per scenario)
B : branching factor
S : seeding factor
$maxLevel$: maximum depth to be explored in the tree
R : retaining factor
Outputs: $outputs$: list of architectures to be shown to the architect

$level \leftarrow 0$
$currentSeeds \leftarrow \{A_0\}$
$outputs \leftarrow \emptyset$
while $level < maxLevel$ **do**
  $candidates \leftarrow \emptyset$
  foreach $dbot \in dbots$ **do**
    foreach $s \in currentSeeds$ **do**
      $candidates \leftarrow generateCandidates(B, S, dbot)$ {Expand each seed via valid tactics}
    end for
  end for
  {Add additional architectures coming from negotiation}
  $candidates \leftarrow candidates \cup doNegotiation(dbots, candidates)$
  {Rank and filter architectures}
  $currentSeeds \leftarrow filterCandidates(S, candidates, filteringCriterion)$
  {Filter resulting architectures to present as output}
  $outputs \leftarrow outputs \cup selectOutputs(R, candidates, outputCriterion)$
end while
return $outputs$

4. Evaluation

To empirically assess the behavior of the combined strategy (and its parameters), we performed a series of experiments with two case studies [Frank 2019]: *ST+* and *CoCoME*, which involve small-to-medium size design spaces. The research questions were the following:

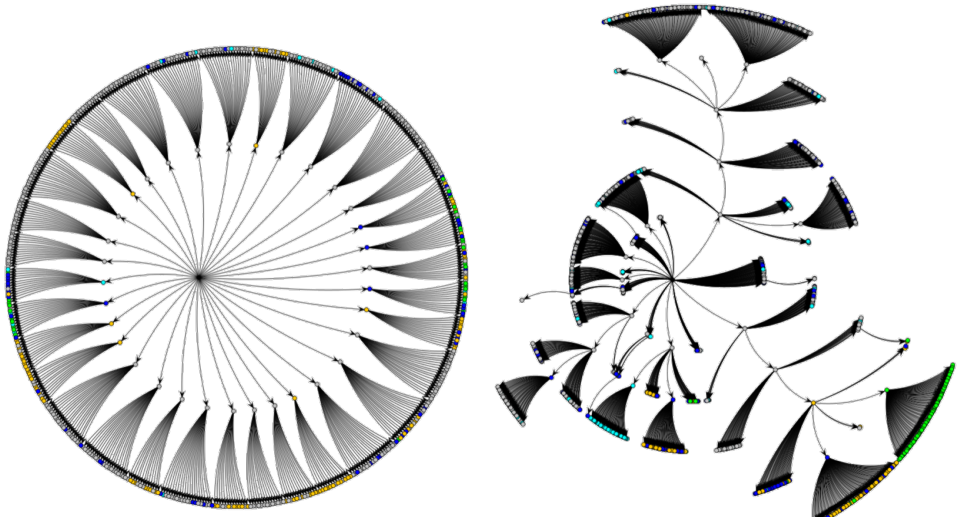
• **RQ1.** How does the *dbots* negotiation affect the number of solutions with satisfying and balanced utility characteristics?

• **RQ2.** What is the role of the branching and seeding factors in reducing the search space?

For each case study, we sampled a large number of candidates (without considering branching or seeding restrictions) up to a fixed search level. For this sampling process, we configured `SQuAT` with *dbots* assigned to performance and modifiability scenarios with their quality thresholds. We set a threshold of ≈ 0.8 for balanced utility. As a result, each dataset comprises a predetermined set of solutions with different characteristics (e.g., satisfying, balanced utility, negotiated, and non-satisfying ones). The two datasets[2], structured as search trees, are shown in Figure 6. The *ST+* tree is balanced, while the *CoCoME* tree has a more irregular structure.

On the (full) tree computed from each dataset, we ran a grid search with alternative configurations and classified the solutions explored by the heuristic according to their characteristics. We were mainly interested in parameters `B` and `S`, but also considered the quality-based and utility-based `filteringCriterion`. For every parameter configuration, we simulated 30 search traces over the tree in order to account for the randomness of the process. An average pruning rate was computed based on the candidates being visited (by the search) in the traces. The pruning

[2]For the sake of reproducibility, the datasets with all the information used in the study are accessible at `http://bit.ly/3RfpOY5`.



(a) ST+ (3 levels) (b) CoCoME (5 levels)

Figure 6. Graphical representations of the search trees for the two case studies.

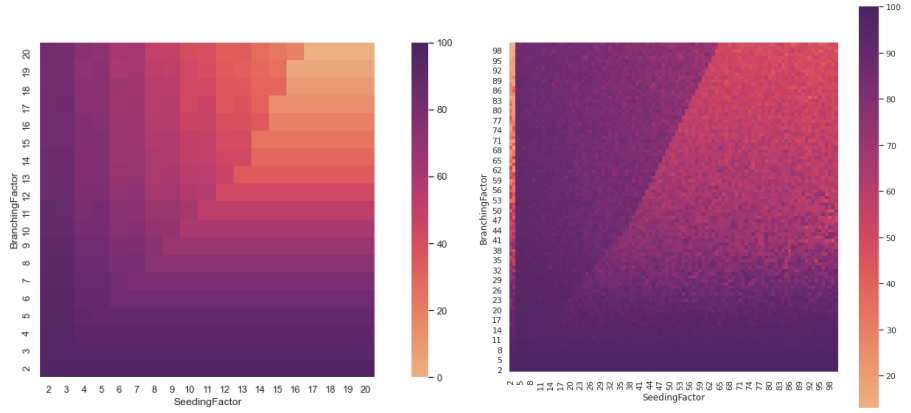
(P) is the ratio of nodes assessed by the *dbots* with respect to the total nodes in the tree (i.e., the dataset). P is an indicator of the computational effort required by the search, recalling that the highest costs come from executing the quality-attribute analyzers on each architecture candidate.

Case study 1 (ST+). The *Extended Simple Tactics* system[3] was partially introduced in Section 2 [Koziolek 2013, Rago et al. 2017]. The PCM specification comprises 9 component types. For optimizing the initial architecture, we set two scenarios for performance and two for modifiability. The search tree contained 554 candidates, including: 154 satisfying solutions (28%), 75 negotiated ones (13.5%) , and 76 solutions with balanced utility (14%). The number of candidates in the sampled space increased significantly with each level, exhibiting a fair distribution of architectures with different characteristics.

We ran search simulations with `B` and `S` in the range $[2-50]$, based on the maximum number of children per node. The `P` values obtained for these configurations are shown in Figure 7a. The darker parts of the heatmap correspond to a more aggressive tree pruning (i.e., high `P` values), which means a faster execution time for the search. As expected, increasing `B` or `S` makes the heuristic explore more candidates, and `P` decreases accordingly. Both `B` and `S` have approximately the same influence on `P`, which can be attributed to the balanced tree structure of *ST+*.

The violin plots in Figure 8a show the kinds of solutions generated by the heuristic when considering pure search or adding negotiation (for all the $P \times B$ configurations). When the *dbots* engaged in negotiation, we observed a higher number of satisfying (*Satisfying-QA*) solutions ($\approx 10\%$) as well as a higher number of solutions ($\approx 20\%$) with a balanced utility (*Balanced-QA*). The improvements were smaller ($\approx 3\%$ and $\approx 10\%$) when using the utility-based criterion (*Satisfying-U* and *Balanced-U*, respectively). We performed a paired samples test (with $\alpha = 0.05$) that confirmed the statistical significance of these differences. We argue that the negotiation is able to find candidates that are not necessarily satisfying but close enough to the thresholds, which cannot be easily detected when the *dbots* try to minimize their quality values. Alternatively, when using the utility-based criterion, the *dbots* seek to maximize utility values; thus, increasing the chances of getting candidates with balanced utility along the way. When checking for differences between pure search and negotiation for low pruning rates, the positive effects of negotiation were attenuated, and both strategies retrieved the same number of solutions.

[3]The *ST+* system specification is publicly available at: `http://bit.ly/3JGFWjV`



(a) ST+ (b) CoCoME

Figure 7. Tree pruning (P) for different branching (*B*) and seeding (S) factors.

(a) ST+ (b) CoCoME

Figure 8. Distribution of the number of solutions with and without negotiation.

Case study 2 (CoCoME). The *Common Community Modeling Example* [Herold et al. 2007] is a benchmark for model-based approaches. It specifies the trading system of a supermarket with many requirements and components. *CoCoME* is a much larger and more realistic model than *ST+*, with a PCM specification[4] having 55 component types. For optimizing the initial architecture, we set twice as many *dbots* as for *ST+*, with four performance scenarios and four modifiability scenarios. The search tree contained 1238 candidates, including: 282 satisfying solutions (23%) , 175 negotiated ones (14%), and 241 solutions with balanced utility (19%) . Unlike the *ST+* design space, the *CoCoME* space had a "wicked" structure with an uneven distribution of the different kinds of solutions. This happened because not all the transformations tried by the *dbots* led to valid architectures, which is a common situation in complex models.

For *CoCoME*, we ran search simulations with B and S in the range $[2-100]$. The P values obtained for these configurations are shown in Figure 7b. The heatmap reveals a similar pattern as for *ST+*, although with higher pruning rates due to the size and shape of the *CoCoME* tree. Figure 8a shows the kinds of solutions obtained with pure search or adding negotiation. There is a small difference in favor of the negotiation for the number of satisfying solutions ($\approx$ 8%) and almost no difference with respect to the balanced solutions, either using a quality-based or a

[4]The *CoCoME* system specification is publicly available at `http://bit.ly/3JGFWjV`

utility-based criterion. The paired samples tests (with $\alpha = 0.05$) only revealed statistical differences for combinations of P and S in specific ranges, although no clear patterns were identified. For instance, negotiation showed higher numbers for satisfying solutions with S in $[65 - 100]$ (irrespective of B) and for balanced solutions with B in $[2 - 10]$ (irrespective of S). When comparing this trend to the more uniform trend in *ST+*, we think the design space complexity is a causal factor since many satisfying candidates were located in the leaves (level 5) of particular sub-trees. Thus, the *dbots* had trouble finding the right paths to those candidates, even if a few intermediate nodes (for those paths) were obtained via negotiation. Certainly, more experiments and other case studies should be analyzed in order to make stronger claims.

Overall, we observed that the proposed heuristic can reduce the computational overhead due to exploring a large number of candidates during the search, making some compromises in the quality of the output. The usage of negotiation seems to improve the tradeoffs of the solutions in terms of meeting the scenario thresholds or having a balanced utility. However, this potential benefit depends on the structure of the search tree, as evidenced by the case studies. An insight from the experiments is that the desired values of branching and seeding should be properly chosen, ideally in a mid-range with respect to the (estimated) number of children per node in the tree.

5. Conclusions and Future Work

In this work, we have presented and assessed a parameterized search heuristic for architecture optimization in SQuAT, which combines negotiation and pruning mechanisms in order to strike a balance between improving the quality tradeoffs of the solutions and reducing the computational costs of the search. The core assumption of our work is that stakeholders aim for balanced solutions, which translates to candidates on which the *dbots* can reach a utility consensus. An initial assessment with two case studies has shown that negotiation can be effective, particularly when limiting the exploration parameters. Beyond SQuAT, we believe that the mechanisms and insights discussed in the paper can be helpful to other architecture optimization approaches, as they often face similar challenges with respect to the effectiveness of the search process.

As discussed in the evaluation, the characteristics of the design space being explored (including the target scenarios) can affect the effectiveness of the proposed heuristic. Unfortunately, such characteristics are difficult to estimate in advance, that is, when configuring the *dbots*. Nonetheless, it should be possible to adapt the search parameters of SQuAT dynamically, as new information (about the space) becomes available [Hoos 2012]. To overcome performance and scalability issues, we plan to investigate the usage of surrogates [Jiang et al. 2020] and parallel-distributed mechanisms, given the multi-agent foundations of SQuAT. This would allow us to apply the approach to larger, more complex architectural models. Along this line, we expect to run SQuAT on some case studies used by related approaches, like *GATSE* and *EASIER*, and compare the performance of our heuristic with that of evolutionary algorithms.

Another interesting mechanism to direct the search or improve its effectiveness is to include human feedback (e.g., regarding architectural preferences) as part of the optimization process. Some studies with evolutionary algorithms have been reported [Quesada et al. 2018]. In SQuAT, currently the architect's preferences (e.g., the utility function, the negotiation policies) have to be entered before the optimization takes place. In future work, we will explore mechanisms to make the *dbots* search process more interactive with the architect, and provide some explanations of the decisions being made in the search, so that the architect can understand and have more confidence in the results of the optimization process.

References

Aleti, A., Buhnova, B., Grunske, L., Koziolek, A., and Meedeniya, I. (2013). Software architecture optimization methods: A systematic literature review. *IEEE Trans. on Soft. Eng.*, 39(5):658–683.

Arcelli, D., Cortellessa, V., D'Emidio, M., and Di Pompeo, D. (2018). Easier: An evolutionary approach for multi-objective software architecture refactoring. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 105–10509.

Bachmann, F., Bass, L., Klein, M., and Shelton, C. (2005). Designing software architectures to achieve quality attribute requirements. *IEE Proceedings - Software*, 152:153–165(12).

Bass, L., Clements, P., and Kazman, R. (2021). *Software Architecture in Practice*. Addison-Wesley Professional, 4th edition.

Becker, S., Koziolek, H., and Reussner, R. (2009). The palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1).

Di Pompeo, D. and Tucci, M. (2023). Quality attributes optimization of software architecture: Research challenges and directions.

Endriss, U. (2006). Monotonic concession protocols for multilateral negotiation. In *5th International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 392–399. ACM.

Fatima, S., Wooldridge, M., and Jennings, N. (2004). An agenda-based framework for multi-issue negotiation. *Artificial Intelligence*, 152(1):1–45.

Frank, S. (2019). Techniques for visualization and interaction in software architecture optimization. Master's thesis, University of Stuttgart.

Herold, S., Klus, H., Welsch, Y., Deiters, C., Rausch, A., Reussner, R. H., Krogmann, K., Koziolek, H., Mirandola, R., Hummel, B., Meisinger, M., and Pfaller, C. (2007). Cocome - the common component modeling example. In *The Common Component Modeling Example: Comparing Software Component Models [result from the Dagstuhl research seminar for CoCoME, August 1-3, 2007]*, volume 5153 of *Lecture Notes in Computer Science*, pages 16–53. Springer.

Hoos, H. H. (2012). *Automated Algorithm Configuration and Parameter Tuning*, pages 37–71. Springer Berlin Heidelberg, Berlin, Heidelberg.

Jiang, P., Zhou, Q., and Shao, X. (2020). *Surrogate Model-Based Engineering Design and Optimization*.

Koziolek, A. (2013). *Automated Improvement of Software Architecture Models for Performance and Other Quality Attributes*. PhD thesis, KIT.

Koziolek, H. and Reussner, R. H. (2008). A model transformation from the palladio component model to layered queueing networks. In Kounev, S., Gorton, I., and Sachs, K., editors, *Proceedings of the SPEC International Performance Evaluation Workshop (SIPEW 2008)*, volume 5119 of *Lecture Notes in Computer Science*, pages 58–78. Springer.

Monteserin, A., Díaz Pace, J. A., Gatti, I., and Schiaffino, S. N. (2017). Agent negotiation techniques for improving quality-attribute architectural tradeoffs. In *Proc. PAAMS '17*, pages 183–195.

Pace, J. A. D. and Campo, M. R. (2008). Exploring alternative software architecture designs: A planning perspective. *IEEE Intell. Syst.*, 23(5):66–77.

Procter, S. and Wrage, L. (2019). Guided architecture trade space exploration: Fusing model based engineering amp; design by shopping. In *2019 ACM/IEEE 22nd Int. Conf. on Model Driven Engineering. Languages and Systems (MODELS)*, pages 117–127.

Quesada, A. R., Romero, J. R., and Ventura, S. (2018). Interactive multi-objective evolutionary optimization of software architectures. *Inf. Sci.*, 463-464:92–109.

Rago, A., Vidal, S. A., Diaz-Pace, J. A., Frank, S., and van Hoorn, A. (2017). Distributed quality-attribute optimization of software architectures. In *Proceedings of the 11th Brazilian Symposium on Software Components, Architectures and Reuse, Fortaleza, CE, Brazil, September 18 - 19, 2017*, pages 7:1–7:10.

Rekha, V. S. and Muccini, H. (2018). Group decision-making in software architecture: A study on industrial practices. *Information and Software Technology*, 101:51 – 63.

Stammel, J. and Reussner, R. (2009). Kamp: Karlsruhe architectural maintainability prediction. In Engels, G., Reussner, R., Momm, C., and Sauer, S., editors, *1st. Workshop des GI-Arbeitskreises Langlebige Softwaresysteme (L2S2): "Design for Future - Langlebige Softwaresysteme"*, pages 87–98.