Quality-driven Multi-objective Optimization of Software Architecture Design: Method, Tool, and Application
Etemadi Idgahi, R.

Citation

Version: Corrected Publisher’s Version
License: Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from: https://hdl.handle.net/1887/30105

Note: To cite this publication please use the final published version (if applicable).
The handle http://hdl.handle.net/1887/30105 holds various files of this Leiden University dissertation.

**Author:** Etemadi Idgahi (Etamaadi), Ramin  
**Title:** Quality-driven multi-objective optimization of software architecture design: method, tool, and application  
**Issue Date:** 2014-12-11
Heuristic-based Application of Search Operators

To make the optimization process faster, we are going to introduce problem-specific search operators. Such a search operator exploits knowledge about the problem domain to change the candidate solutions into one that is expected to be an improvement. This chapter proposes an answer to RQ3 defined in Section 1.2:

In which ways can meta-heuristic optimization be improved in order to make the process of reaching optimal architectural solutions faster?

According to the literature, it is known that using problem-specific operators can be beneficial for optimization approaches [ABG+14] [TK11]. However, because we are considering multiple objectives for architecture optimization, new challenges rise. The first challenge is that a heuristic technique usually improves only one specific quality attribute and as a result it may deteriorate other objectives. The second challenge is that in multi-objective optimization problems (more than 3 objectives) comparing the results of two optimization processes is difficult because the solutions are mostly non-dominated compared to each other. So, it is not trivial to figure out what is the best way of combining the heuristic-based search operators for multiple objectives. In this chapter, an experiment was set up to compare various combinations of heuristic-based search operators for an embedded system architecture problem with four objectives based on a optimality-measure called ‘Averaged Hausdorff distance’.

This chapter is structured as follows. Section 7.1 introduces the idea of problem-specific search operators in general and then some of their examples for the software architecture optimization problem specifically. After that, in Section 7.2 we discuss the use of architectural patterns and anti-patterns in heuristic-based search operators to try
to reach the optimal solution faster. Section 7.3 introduces different ways of combining the aforementioned search operators in the optimization algorithm. Section 7.4 shows the results of an experiment based on a real-world case study and discusses the effects of these combination of the evolutionary algorithm. Lastly, Section 7.5 summarizes the chapter.

7.1 Problem-Specific Search Operators

Evolutionary algorithms are often the method of choice for solving optimization problems with non-standard representations of the candidate solutions (e.g. special types of graphs). When applying EA to such non-standard domains, in general two approaches can be distinguished: (i) Introduce a genotype-phenotype mapping to a canonical representation, e.g. bit-string or continuous vectors on which standard operations, such as one-point crossover or bit mutation can be performed. (ii) Perform the search directly on the phenotype space and formulate problem-specific mutation and recombination operators as transformations of solutions in the phenotype space. While for the first approach out-of-the-box implementations of EA can be used after the genotype-phenotype mapping has been established, the second approach requires the formulation of new initialization, mutation and recombination operators. Often this effort is rewarded by a (much) higher performance of the EA as comparative studies in various domains show, for instance in chemical process design [STT+08] and decision diagram design [DW00].

In [EGS01] mutation and recombination operators on graphs that represent chemical engineering process flow sheets were introduced in the form of graph rewriting rules. These rules define patterns in the flowsheet and how these patterns can be replaced by alternative patterns with a similar function. As opposed to operators that work with standard representations, this problem specific approach makes it easy to define transitions that lead from feasible structures to new feasible structures. Such problem-specific operators have a relatively high probability of finding improvements. Similar graph-based EA were successfully applied in other domains such as analogue circuit design [NHAH04]. In software architecture design such direct representations with tailored operators have not yet been applied.

In the following, we explore some possible problem-specific search operators for the problem of software architecture design:

7.1.1 Caching for improving performance

Basically, there are two main caching strategies that can be described as patterns [Rot06]: Primed Cache and Demand Cache. If the data required for performing a certain computation is known prior to the start of that computation, then the system
can store it before the computation starts, which is called Primed Cache. However, in case the data required by a computation can vary for each run, the system can bring the data into the memory whenever required and keep it for future use. This strategy is called the Demand Cache pattern.

Figure 7.1 depicts a search operator that transformation an architecture fragment by introducing a caching-pattern. In this pattern CompB is replaced by a combination of Cache component and CompB. So, instead of directly calling of CompB by CompA, CompA calls the Cache component and then, only if needed, it calls CompB.

7.1.2 Voter pattern for improving safety

Masking faults is one of the primary approaches to improve the behaviour of a system in a faulty environment. N-modular redundancy and N-version programming are
well-known fault masking methods. These approaches use redundant modules and a voting unit to hide the occurrence of errors. The voter arbitrates between the achieved results and produces a single output [LSBB03].

Figure 7.2 depicts the introduction of a voter by means of a transformation (which can be implemented as a search operator) with three software replicas. It shows that CompB is replaced by a combination of three replicated CompB’s and one Voter component. Now, instead of directly calling of CompB directly from CompA, Voter collects the votes from each of the three CompB’s, and then forwards the majority answer to CompA.

7.1.3 Encryption/Decryption for improving security

Encryption provides message confidentiality by transforming readable data (plain text) into an unreadable format (cipher text). That unreadable cipher text can be understood only by the intended receiver after a process called Decryption. Decryption makes the encrypted information readable again.

Figure 7.3 shows this search operator’s transformation. In this CompA is replaced by a combination of an Encryptor component and a CompA. Also CompB is replaced by a combination of Decryptor component and CompB. So, CompA and CompB can communicate securely.

7.2 Heuristic-based Search Operators by Anti-patterns

Architecture- and design-patterns capture expert knowledge about "best practices" in software design by documenting general solutions that may be customized for a particular context. They make it possible to reuse the knowledge of software design and to focus on quality attributes such as performance. Software anti-patterns are conceptually similar to patterns in that they document recurring solutions to common design problems (i.e. the "bad practices") as well as their solutions: what to avoid and how to solve the problems [CMT10].
Software architecture design patterns look at the positive and constructive features of a software system, and suggest common solutions. In contrast, anti-patterns look at the negative and destructive features of a software system, and present common solutions to the problems that make negative consequences. Bottlenecks affect quality attributes negatively. Therefore we explore an approach that first detects anti-patterns and used these as indicators of possible bottlenecks in architectural solutions. Next, a suitable transformation needs to be applied to remove/reduce the suspected bottleneck. In our study we consider four architecture heuristics as problem-specific search operators. The first two operators are derived from the Concurrent Processing Systems anti-pattern. As it is stated in [Tru11], ”[This anti-pattern] occurs when processing cannot make use of available processors”. In other words, the processes running on the system cannot use the available resources effectively. This could happen when the processes are assigned to the processors in a non-balanced way. The other operators address other quality attributes: cost and reliability. We explain these in more detail in the subsequent sections.

7.2.1 Search Operator: Component Movement

According to Concurrent Processing Systems anti-pattern, non-balanced assignment of processes to processors can make the system slow and cause a performance bottleneck. This operator moves the most resource-intensive component deployed on the highest utilized processor to the least utilized processor in the architecture.

Figure 7.4 shows a example system with four nodes. “t” represents execution time of each software component on the deployed node. As can be seen, there may be a node (node 1) in a software system containing many components which cause a high utilization. On the other hand, there is another node (node 4) with just one component and low utilization. Therefore, as a whole, the collection of resources is not used efficiently.

This anti-pattern suggests a rearrangement of allocating components to available resources. A more balanced allocation of the components to the nodes, after applying this anti-pattern, is illustrated in Figure 7.5.

7.2.2 Processor Change for Performance

When there is a processor with high utilization in the architecture, a solution to reduce utilization is replacing it with a processor more processing power. In AQOSA, there is a repository of available hardware resources. A processor with higher clock rate can reduce the overall utilization of the system, so it can be selected for replacement.
7.2.3 Processor Change for Cost

Cost is another quality attribute and often important optimization objective. Replacing a processor with high clock rates are often more expensive. Hence using these to improve utilization often deteriorates the cost objective. Conversely, this operator replaces the less utilized processors with cheaper ones thereby reduces the cost dimension.
7.2.4 Processor Change for Reliability

This operator is designed to decrease the probability of failure and consequently increase reliability. There may be some processors in an architectural solution which have a high probability of failure. They should be identified and replaced by the processors with lower probability of failure.

7.3 Combining Search Operators in the Genetic Algorithm

The heuristic search operators we presented in the previous sections are targeted for improving one particular quality attribute. However, the same operator might have no effect or might even deteriorate other quality properties. For example, the ‘Component Movement’ operator is beneficial for response time and the ‘Processor Change for Performance’ operator is beneficial for processor utilization while they are not useful for cost and failure probability. The operators ‘Processor Change for Cost’ and ‘Processor Change for Reliability’ act the same way in favour of different objectives.

In order to use these directed search operators in genetic optimization, we must find some way of using these operators such that a sufficiently broad area of the search space is covered – so that not one dimension of the objective-function is favoured over others. Moreover, in order to maintain the important random-aspect of genetic algorithms, we should find a way in which to combine the directed search operators with generic GA operators such as mutation and cross-over. In this chapter, the extent to which heuristic-based search operators can improve multi-objective optimization of software architecture is studied.

For the experiments in this study, the mating procedure of GA has been modified as follows: Two parents are needed to be operated on by the search operators and they generate two offsprings – that are transformed in different ways. Invoking the search operators could be done in various orders which is called ‘Combinations’. So, assume we have this set of operators:

\[
\text{Operators' set} = \{ 'swMove', 'pc4Perf', 'pc4Cost', 'pc4Rely' \} \quad (7.1)
\]

Then, we have one function which return generic operators, and two functions to choose among these operators, which return one operator on each call:

\[
\text{GE()} : \text{return generic GA operator.} \quad (7.2)
\]

\[
\text{RA(set)} : \text{return random operator out of the set.} \quad (7.3)
\]

\[
\text{RR(set)} : \text{return an operator in round-robin order out of the set.} \quad (7.4)
\]
Finally, for generating offspring we need to call two operators which are chosen by one of aforementioned functions:

\[
\text{Offspring}(\text{Parents}) = \{\text{Child}_1, \text{Child}_2\}, \quad \text{where:} \quad \text{Child}_1 = \text{Operator}_1(\text{Parent}), \quad \text{Child}_2 = \text{Operator}_2(\text{Parent}).
\]  

(7.5)

The following combinations of operators are considered for invoking search operators to act on a pair of parents and to generate two offsprings for the next generation:

### 7.3.1 Random

For both offsprings, the mating procedure completely randomly selects heuristic-based operators. Figure 7.6 depicts this combination of anti-patterns. In other words:

\[
\text{Operator}_1 = \text{RA}(\text{set}), \quad \text{Operator}_2 = \text{RA}(\text{set}).
\]

(7.6)

![Random combination of heuristic-based search operators](figure76.png)

**Figure 7.6:** Random combination of heuristic-based search operators

### 7.3.2 Sequential

For both offsprings, the mating procedure picks heuristic-based operators sequentially. It means that it uses the round robin ordering for operators. It is depicted in Figure 7.7. In other words:

\[
\text{Operator}_1 = \text{RR}(\text{set}), \quad \text{Operator}_2 = \text{RR}(\text{set}).
\]

(7.7)

![Round-robin choice of operator](figure77.png)

**Figure 7.7:** Sequential combination of heuristic-based search operators
7.3.3 Random-Sequential

For one offspring, the mating procedure picks a heuristic-based operator in the random order, and for the other one, it picks the operator sequentially (As depicted in Figure 7.8). In other words:

\[
\text{Operator1} = RA(\text{set}), \quad \text{Operator2} = RR(\text{set}).
\]  

(7.8)

![Figure 7.8: Random-Sequential combination of heuristic-based search operators](image)

7.3.4 Half-Random

For one offspring, the mating procedure picks a heuristic-based operator randomly, and for the other one, it uses the generic operators (Crossover and/or Mutate). Figure 7.9 depicts this combination of operators. In other words:

\[
\text{Operator1} = GE(), \quad \text{Operator2} = RA(\text{set}).
\]  

(7.9)

![Figure 7.9: Half-Random combination of heuristic-based search operators](image)

7.3.5 Half-Sequential

For one offspring, the mating procedure picks a heuristic-based operator in round robin order. For another one, it picks a generic operator (As depicted in Figure 7.10). In other words:

\[
\text{Operator1} = GE(), \quad \text{Operator2} = RR(\text{set}).
\]  

(7.10)
Figure 7.10: Half-Sequential combination of heuristic-based search operators

7.3.6 Half-Random-Sequential

For one offspring, the mating procedure picks the generic operators (Crossover and/or Mutate), and for the other one, it switches between random and sequential ordering from generation to generation. This combination is depicted in Figure 7.11. In other words:

\[
\text{Operator}_1 = \text{GE}(\cdot), \quad \text{Operator}_2 = \begin{cases} \text{RA}(\text{set}) & \text{if generation is even} \vspace{0.1cm} \\
\text{RR}(\text{set}) & \text{if generation is odd} \end{cases} \tag{7.11}
\]

Figure 7.11: Half-Random-Sequential combination of heuristic-based search operators

7.4 Experiment

To compare the aforementioned strategies for combining operators, an experiment was performed using the SAAB Instrument Cluster case study (which was discussed in Section 5.3). The system represents the Saab 9-5 Instrument Cluster Module ECU (Electronic Control Unit, a node in a network) and the surrounding sub-systems. The Instrument Cluster Module is responsible for 8 concurrent user functions. For providing these functionalities, it should be able to handle 6 sporadic tasks and 4 periodic tasks concurrently.

The goal of the experiment is to compare the combinations of operators, in terms of the speed by which they find optimal solutions. To this end, we defined an experiment with the following steps:
1. First we try to find out what would be the ideal Pareto front. Given that this may take very long to computer, we use the following as an approximation of this: We run the optimization process with only the generic search operators for a very high number of generations. This gives the algorithm enough opportunity to approximate the ideal Pareto front within a small margin. We used this set of solutions as the reference Pareto front for comparison with other strategies for combining search operators.

2. We run the optimization with generic operators (without heuristic-based search operators) and also with the six strategies for combining the operators (as described in Section 7.3, all of them with a fixed number of generations (each optimization process 20 times). In this situation, better combinations of operators are expected to achieve better results.

3. As a measure of the quality of the Pareto front that was found, we assess the distance between the results from step1 and step2. A smaller distance between the Pareto fronts, means that the combination of operators used in step2 achieve better results.

7.4.1 Experiment Setup

For step1, we run the optimization with the following parameter settings: number of generations=200, initial population size(\(\alpha\))=1000, parent population size (\(\mu\))=250, number of offspring(\(\lambda\))=500, archive size=50, crossover rate is set to 0.95.

For the step2, we run the optimization 20 times for each combination strategy with the following settings: number of generations=15, initial population size(\(\alpha\))=100, parent population size (\(\mu\))=25, number of offspring(\(\lambda\))=50, archive size=20, heuristic rate and crossover rate are both set to 0.95.

At the step3, to calculate the distance between two sets of Pareto fronts (that result from step1 and step2) we used a measure called ‘Averaged Hausdorff distance’. Schütze et al. [SELC12] defined ‘Averaged Hausdorff distance’ as:

\[
\max \left( \left( \frac{1}{N} \sum_{i=1}^{N} \text{dist}(x_i, Y)^p \right)^{1/p}, \left( \frac{1}{M} \sum_{i=1}^{M} \text{dist}(y_i, X)^p \right)^{1/p} \right)
\]  

(7.12)

Where \(X = x_1, x_2, ..., x_n\) and \(Y = y_1, y_2, ..., y_m\) are two Pareto fronts with sizes of \(N\) and \(M\) respectively. We set \(p = 1\) for this experiment.

For generating architectural solutions, the set of software components is given by the SAAB instrumentation cluster software architecture. Note that these components may be replicated in any actual architecture solution. In addition, the repository contains the following hardware components that may be used:
• 28 Processors: ranging over 14 various processing speeds from 66MHz to 500MHz; Each of them has two levels of failure rate. A processor is more expensive if it has less chance of failure and vice versa.

• 4 Buses: with bandwidths of 10, 33, 125, and 500 kbps, and latencies of 50, 16, 8, and 2 ms. A bus is more expensive if it supports higher bandwidth.

7.4.2 Experiment Results

Figure 7.12 depicts the differences between the results of optimizations without (gray box) and with (white boxes) heuristic-based search operators. It shows the boxplot chart of the distances for each combination of operators (runs of 20 iterations) as described in Section 7.3 and the reference Pareto front obtained in step1. In the chart, lower values indicate a better combination strategy because it is closer to the optimal results. For calculating the distance between Pareto fronts, we normalized the values of four dimensions and then we used Equation 7.12 to calculate the averaged Hausdorff distance of two Pareto fronts. Therefore, the vertical axis in Figure 7.12 represents the averaged Hausdorff distance.

Figure 7.12: Averaged Hausdorff Distance of different operator combinations
The plots in Figure 7.12 show that combinations with one generic operator generated offspring (Half-*) cause boxplots with a higher spread, or in other words, they are more dependent on luck for finding optimal results. They are more similar to the results of the optimization with only generic operators. Instead, combinations with tighter boxplots represent better combinations: independent of the randomness in the algorithm, they are more likely to find better solutions. Among these latter ones, the **Sequential** and **Random-Sequential** combinations show lower median values and tight boxes, hence perform the best.

### 7.5 Summary

In this chapter, (i) the usefulness of problem-specific operators in the software architecture domain was discussed, and (ii) a comparison between various approaches for combinations of heuristic-based search operators was performed. To do so, knowledge of architecture anti-patterns was implemented by means of problem-specific search operators within an evolutionary algorithm. The case study experiment in this chapter was defined based on a real world case study and was applied to a 4-objective software architecture optimization problem. The results of the experiment showed that search operators for improving one objective can be used in multi-objective optimization context. The results indicated that proper combination strategies for heuristic-based search operators can lead optimization algorithms to optimal solutions faster. However, for preventing not getting trapped in suboptimal solutions, room for randomness should always be accounted for in the optimization (esp. offspring/mating) process.

As future work, it will be interesting to study situations with unbalanced number of operators in which each operator is forcing specific objective. For example, 3 operators in favour of one objective and 2 operators in favour of conflicting objectives. Also, another topic for future work can be studying effects of weighting heuristic-based search operators on the results of the optimization process.