Combinations of Antipattern Heuristics in Software Architecture Optimization for Embedded Systems

Ramin Etemaadi[1] and Michel R.V. Chaudron[2,1]

[1] Leiden Institute of Advanced Computer Science, Leiden University, Netherlands
[2] Joint Department of Computer Science and Engineering, Chalmers University of Technology and Gothenborg University, Sweden
`etemaadi@liacs.nl` and `chaudron@chalmers.se`

Abstract. A large number of quality properties need to be addressed in nowadays complex embedded systems by architects. Evolutionary algorithms can help architects to find optimal solutions which meet these conflicting quality attributes. Also, architectural patterns and antipatterns give the architect knowledge of solving design bottlenecks. Hence, antipatterns heuristics have been used as domain-specific search operators within the evolutionary optimization. However, these heuristics usually improve only one quality attribute and using them in multiobjective problem is challenging. This paper studies the extent to which heuristic-based search operators can improve multiobjective optimization of software architecture for embedded systems. It compares various combinations of heuristic-based operators in a real world automotive system case study.

Keywords: Embedded System Architecture Design Optimization; Architectural Antipatterns; Domain-Specific Search Operators; Evolutionary Multiobjective Optimization (EMO);

1 Introduction

The architecture has deep impact on non-functional properties of a system such as performance, safety, reliability, security, energy consumption and cost. Due to the complexity of today's software systems, designing a system which meets all its quality requirements becomes increasingly complex. Hence, system architects have to employ optimization techniques to be able to explore more design possibilities and to find optimal architectural solutions. Metaheuristic approaches frame the challenge of designing architectures as an optimization problem and iteratively try to improve a candidate solution with regard to the given quality attributes. Evolutionary Algorithms (EA), as a well-known metaheuristic approach, is a common optimization technique for solving system architectural problems. However, EA for generating new solutions uses generic search operators, such as *Crossover* or *Mutate*, which are blind to the problem and do not take into account the domain knowledge. To overcome this issue, domain-specific search operators have been proposed. The downside of using domain-specific search operators, is that the algorithm might find local optimal solutions. In addition, each domain-specific operator is usually useful only for one specific objective, and this is a threat to the optimality of results in multiobjective problems.

This paper studies and compares various combinations of EA search operators (both domain-specific and generic) for multiobjective optimization of software architecture. The domain-specific operators are motivated by software architectural antipatterns. However, each heuristic-based search operator improves only one quality attribute of the solution, which is challenging for multiobjective problems. We apply various combination of operators to a case study, which is derived from a real world automotive embedded system.

The paper is organized as follows: Firstly, Section 2 discusses related work. Then, Section 3 describes our optimization framework, how we define heuristic-based search operators for optimization of software architecture, and the proposed combinations of these heuristic-based search operators. The case study that we applied our approach on, is represented in Section 4. Finally, The paper concludes in Section 5.

2 Related Work

In the following, the state of the art of approaches which employ software design heuristics for optimizing architecture design are discussed:

2.1 PerOpteryx

Koziolek et al. [1] introduced a hybrid approach that incorporates architectural performance "Tactics" into an evolutionary optimization process. They defined architectural tactics to improve two quality attributes: Performance and Cost. They implemented those tactics as part of the evolutionary optimization process. They showed that by using tactics, optimization algorithms can achieve better solutions. However, their experiment was conducted in information systems context and with 2-dimension optimization settings.

2.2 Antipatterns in Palladio

Turbiani et al. [2] discussed the advantages of using software performance antipatterns in an iterative manner. They introduced a couple of performance antipatterns and defined automatic approach to detect and solve the bottlenecks in software architecture solution. They demonstrated by applying this technique iteratively, the system performance can be improved significantly. However, they did not discuss quality attributes other than performance. They also did not integrate their approach within an evolutionary optimization process. Thus, the downside of their approach is that without having generic degrees of freedom and involving randomness in the optimization iterations, the optimality of the results is highly dependent on the initial architecture.

3 Multiobjective Optimization of Software Architecture

According to the studies in the related work, it is known that using domain-specific operators is beneficial for software architecture optimization. However, by increasing the number of objectives for architecture optimization, we face new challenges. The first challenge is that each heuristic technique usually improves only for one specific quality attribute and as a result it may deteriorate other objectives. The second challenge is that in multiobjective 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 paper, we defined an experiment to compare various combinations of heuristic-based search operators for an embedded system architecture problem with four objectives based on a measurement called *'Averaged Hausdorff distance'*. For this reason, we used a real world case study from automotive industry. Our optimization framework, with heuristic-based operators based on various combinations, was applied on that case study.

In this section, sub-section 3.1 describes brie
y the AQOSA optimizationframework
(The details of the AQOSA framework is reported by authors in [8]).
Then, sub-section 3.2 introduces the heuristic-based search operators which we used in this experiment. Subsequently, sub-section 3.3 discusses various approaches for combinations of heuristic-based search operators.

3.1 AQOSA Framework

AQOSA is a framework which uses a metaheuristic optimization approach based on generic algorithms for automated software architecture design. The framework supports analysis and optimization of multiple quality attributes including response time, utilization, safety and cost. It uses an architectural Intermediate Representation (IR) model for describing the architectural design problem. The AQOSA framework takes as input:

i) an initial functional part of the system (i.e. components that provide the needed functionality and their communications),

ii) a set of typical usage scenarios (including triggers to create workloads),

iii) an objective function (implying which architecture properties should be optimized),

iv) a repository that contains a set of specifications of hardware and software components.

Then, AQOSA iterates through the following steps:

1. Generate a new set of candidate architecture solutions: Hence, AQOSA uses a representation of the architecture where it knows which are the degrees of freedom in the design and how to generate alternative architecture.
2. Evaluate the new set of candidate architecture solutions for multiple quality properties: This works by generating analysis models from the architecture model using model transformations and then analysing these models.

3. Select a set of optimal solutions. It is based on the chosen evolutionary algorithm.
4. Iterate to step 1 until some stopping criterion holds. This can be a maximum number of generations or a criterion on the objective function.

Below we briefly present the framework modules:

Architecture Modelling Because AQOSA is designed to optimize architectures in a wide range of domains, it aims to be independent of specific modelling languages. Therefore, it uses its own internal architecture representation, AQOSA intermediate representation (AQOSA-IR). The IR-model integrates multiple quality modelling perspectives for the architectural level optimization purpose. However, it is possible to transform well-known architectural models like AADL or UML/MARTE to this intermediate representation.

Architecture Optimization The AQOSA optimizer tries to optimize the software architecture with respect to potentially conflicting quality attributes based on Genetic Algorithm (GA). To this end, it automatically generates new architectural design alternatives. It has been implemented based on the Opt4J optimization framework [3].

Degrees of Freedom When an architect finalizes an architectural design for a system, generally there are still some ways in which the solution can be varied without changing the functionality. We call them *Degrees of Freedom*(DoF). The component-based paradigm that underlies our approach, allows us to recompose components in different topologies without changing the functionality of the system. We support the following degrees of freedom in the AQOSA framework: (1) Number of hardware nodes, (2) Number of connections between hardware nodes, (3) Network topology, (4) Software on hardware allocation, (5) Software components replacement, (6) Processor nodes replacement, (7) Communication lines replacement.

Evolutionary Algorithms AQOSA is compatible with well-known Evolutionary Multi-Objective Algorithms (EMOA). For the experiment of this paper we employed the famous algorithm NSGA-II (proposed by Deb [4]).

Architecture Evaluation The AQOSA evaluation sub-system gets an evaluation model which is transformed from an AQOSA-IR and a decoded genotype for specific evaluation purpose (e.g Response Time or Safety). It feeds these models to each evaluator and returns the results to the optimization module. In this experiment, we evaluated four quality attributes: Response time, processor utilization, safety, and cost. Performance attributes (response time and utilization) have been implemented by extending the JINQS [5] Queuing Networks (QN) library. For safety analysis we have implemented a Fault Tree Analysis (FTA) method introduced by Forster [6].

3.2 Heuristic-based Search Operators

Software architecture design patterns look at the positive and constructive features of a software system, and suggest common solutions. In contrast, antipatterns look at the negative and destructive features of a software system, and present common solutions to the problems that make negative consequences [7]. Because bottlenecks affect quality attributes negatively, we use antipatterns in order to diagnose the bottlenecks in architectural solutions in our optimization approach. Further, we describe four architecture heuristic as domain-specific search operators which we applied to the case study in this paper. The first two operators are derived from *Concurrent Processing Systems* antipattern. As it is stated in [7], "[This antipattern] 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 [7]. The later two operators use the same principle for other quality attributes.

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

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 better processor. 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.

Processor Change for Cost The former operator tackles processor utilization bottleneck. However, cost is another quality attribute and optimization objective. Replacing a processor with higher clock rate (probably more expensive) to solve utilization makes a deterioration for cost objective. Conversely, this operator replaces the less utilized processors with cheaper ones and probably lower clock rate.

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

3.3 Combination of Optimization Operators

Each heuristic-based search operator is useful for the relevant quality attribute that is made for. However, it might have no effect on the other quality properties, or might deteriorate them. For example, the “Component Movement” operator is beneficial for response time and “Processor Change for Performance” operator is beneficial for processor utilization while they are not useful for cost and failure probability. “Processor Change for Cost” and “Processor Change for Reliability” operators act the same way in favour of different objectives. In this paper, the extent to which heuristic-based search operators can improve multiobjective optimization of software architecture is studied.

To define the experiment in this study, we overrode the mating procedure of GA as follows: Two parents are needed to be operated by the search operators and they generate two offsprings. Calling the search operators could be done in various calling orders which we called *’Combinations’*. The following combinations of operators are considered for calling search operators to act on a pair of parents and generate two offsprings for the next generation:

Random For both offsprings, the mating procedure picks heuristic-based operators randomly.

Sequential For both offsprings, the mating procedure picks heuristic-based operators sequentially. It means that it uses the round robin ordering for operators.

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.

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).

Half-Sequential For one offspring, the mating procedure picks a heuristic-based operator in the round robin order, and for the other one, it picks the generic operators (Crossover and/or Mutate).

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.

4 Case Study

4.1 Automotive Subsystem

To compare the aforementioned combinations of operators, we applied them to a real case study from automotive industry. The case study was conducted at Saab Automobile AB and has been reported in the previous authors’ work [8]. The

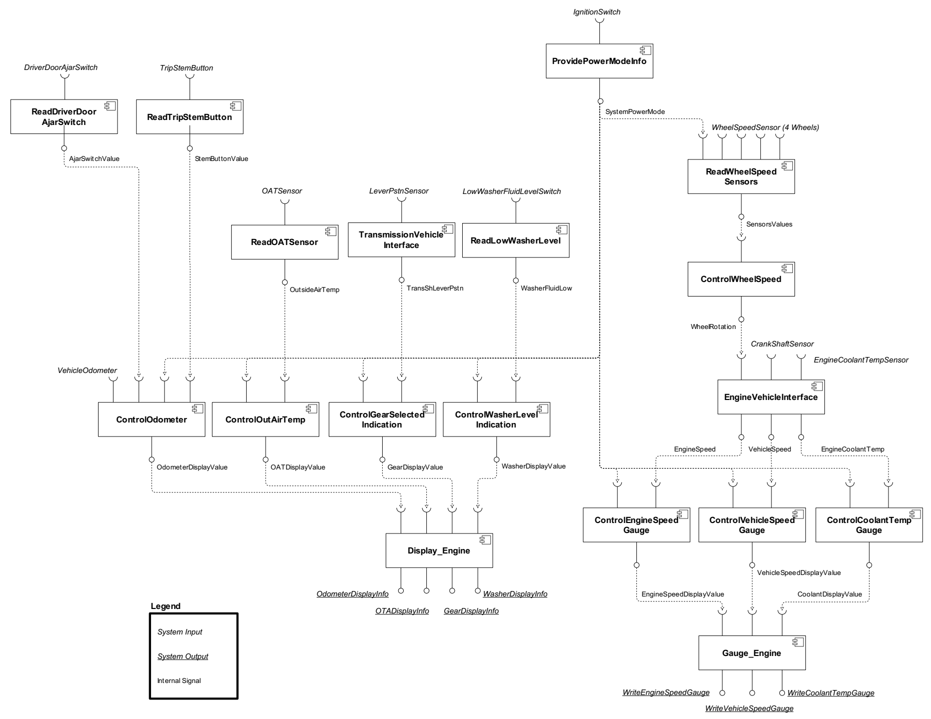

Fig. 1. Component diagram of Automotive Instrument Cluster system.

system represents the Saab 9-5 Instrument Cluster Module ECU (Electronic Control Unit, a node in a network) and the surrounding sub-systems. It consists of 18 components as depicted in Figure 1. The Instrument Cluster Module is responsible for 8 concurrent user functions. Hence, for providing these functionalities, it should be able to response 6 sporadic tasks and 4 periodic tasks concurrently. Details of these tasks have been reported in [8] and [9].

For generating new architectural solutions, the repository of hardware components contains these elements:

– 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.

As an estimate of the size of the design space in this case study, consider the following reasoning: Assume we omit architecture topology changing and fix an architecture with six processors and three bus lines for their interconnections (exactly like the current realization in the industry). For these constraints there are $28^6 \cdot 4^3$ different possibilities, which is more than 30 billion architectures.

When also considering variations in the architecture topologies, this number would be even considerably higher.

4.2 Results

The goal of the experiment is to compare the combinations of operators, in terms of achieving optimal solutions faster. To this end, we defined the experiment with these steps:

1. We run the optimization process with high number of generations. So, with giving enough time to the algorithm, it could achieve optimal solutions. We used this set of solutions as the reference Pareto front in comparison with other Pareto fronts.
2. We run optimization with generic operators (without heuristic-based search operators) and also with six various combinations of optimization operators, all of them with low number of generations (each optimization process 20 times). In this situation, better combination can achieve optimal results within few number of generations.
3. We measured the distance between the results from *step1* and *step2*. Shorter distance between the Pareto fronts, or in other words, closer result from *step2* to the results of *step1* means that combination could achieve better results in few number of generations. We interpret that combination as a better combination.

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

For the *step2*, we run 20 times for each combination with these settings: number of generations=15, initial population size(α)=100, parent population size (μ)=25, number of offspring(λ)=50, archive size=20, heuristic rate and crossover rate are both set to 0.95.

At the *step3*, to calculate the distance between two set of Pareto front results which achieved from *step1* and *step2*, we used a measurement called ‘Averaged Hausdorff distance‘. Schütze et al. [10] defined ‘Averaged Hausdorff distance‘ as:

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

Where $X = x_1, x_2, ..., x_n$ and $Y = y_1, y_2, ..., y_m$ are two Pareto fronts with the size of N and M. We set $p = 1$ for this experiment.

Figure 2 depicts the difference between the results of optimization with (white boxes) and without (gray box) heuristic-based search operators. It shows the boxplot chart of the distance between 20 runs of each combination of operators as described in Section 3.3 and the *step1*. In the chart, lower values indicate better combination because it represents the distance with optimal results. For

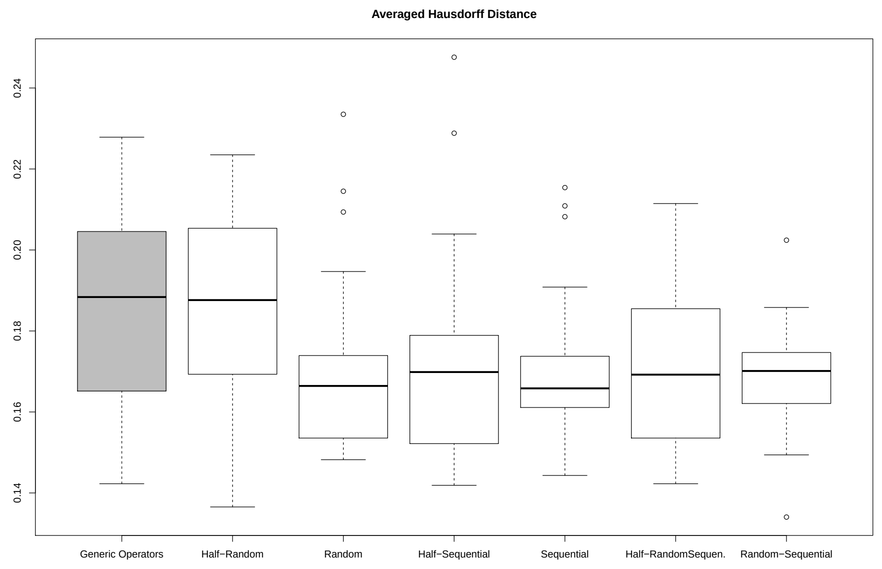

Fig. 2. Averaged Hausdorff Distance of different operator combinations.

calculating the distance between Pareto fronts, we normalized the values of four dimensions and then we used Equation 1 to calculate the averaged Hausdorff distance of two Pareto fronts. Therefore, vertical axis in Figure 2 represents the averaged Hausdorff distance.

The plots in Figure 2 show that combinations with one generic operator generated offspring (Half-*) cause wider boxplots, or in other words, they are more dependent on luck for finding optimal results. They are more similar to the results of running with generic operators. Instead, combinations with tighter boxplots represent better combinations. Among them, *Sequential* and *Random-Sequential* combinations perform best. Because, they show lower median values and tight boxes.

5 Conclusions

In this paper we introduced a comparison between various approaches for combinations of heuristic-based search operators in a model-based tool that integrates multiple quality analysis of software architecture. We implemented knowledge of architecture antipatterns as the domain-specific search operators within an evolutionary algorithm. We defined an experiment based on a real world case study and we applied it for a 4-objective software architecture optimization problem. We showed that search operators for improving one objective can be used in multiobjective optimization context. The results showed that proper combination of heuristic-based search operators can lead optimization algorithm to optimal solutions faster. However, for preventing not trapping in suboptimal solutions, rooms for randomness should always be considered in the optimization parameters settings.

As the future work, it is interesting to study effects of weighting heuristic-based search operators on the results of optimization process, especially when number of operators which forcing each objective are unbalanced.

6 Acknowledgments

This work has been supported by the Netherlands national project OMECA (Optimization of Modular Embedded Computer-vision Architectures).

Also, authors thank Noushin Khaki for her contribution in implementation of these operators for AQOSA framework.

References

1. Koziolek, A., Koziolek, H., Reussner, R.: Peropteryx: automated application of tactics in multi-objective software architecture optimization. In Crnkovic, I., Stafford, J.A., Petriu, D.C., Happe, J., Inverardi, P., eds.: QoSA/ISARCS, ACM (2011) 33–42
2. Trubiani, C., Koziolek, A.: Detection and solution of software performance antipatterns in palladio architectural models. In Kounev, S., Cortellessa, V., Mirandola, R., Lilja, D.J., eds.: ICPE, ACM (2011) 19–30
3. Lukasiewycz, M., Glaß, M., Reimann, F., Teich, J.: Opt4J: a modular framework for meta-heuristic optimization. In Krasnogor, N., Lanzi, P.L., eds.: GECCO, ACM (2011) 1723–1730
4. Deb, K., Agrawal, S., Pratap, A., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: Nsga-ii. IEEE Transactions on Evolutionary Computation **6**(2) (2002) 182–197
5. Field, T.: JINQS: An Extensible Library for Simulating Multiclass Queueing Networks. (2010)
6. Förster, M., Trapp, M.: Fault Tree Analysis of Software-Controlled Component Systems Based on Second-Order Probabilities. In: ISSRE, IEEE Computer Society (2009) 146–154
7. Trubiani, C.: Automated generation of architectural feedback from software performance analysis results. PhD thesis, Universita di L'Aquila (2011)
8. Etemaadi, R., Lind, K., Heldal, R., Chaudron, M.R.V.: Quality-driven optimization of system architecture: Industrial case study on an automotive sub-system. Journal of Systems and Software **86**(10) (2013) 2559–2573
9. Etemaadi, R., Lind, K., Heldal, R., Chaudron, M.R.V.: Details of an Automotive Sub-System: Saab Instrument Cluster Module. Technical report, number: 2013-01, Leiden Institute of Advanced Computer Science, Leiden University (2013)
10. Schütze, O., Esquivel, X., Lara, A., Coello, C.A.C.: Using the averaged hausdorff distance as a performance measure in evolutionary multiobjective optimization. IEEE Trans. Evolutionary Computation **16**(4) (2012) 504–522