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 (Etmaadi), Ramin  
**Title:** Quality-driven multi-objective optimization of software architecture design: method, tool, and application  
**Issue Date:** 2014-12-11
Chapter 8

Software Architecture Optimization for Software Product Lines

This chapter proposes a solution for RQ4 which is defined in Section 1.2:

In what aspects can search-based approaches improve the process of designing a software architecture for a family of products in a software product line?

In this chapter, a new search-based approach for generating a set of optimal software architectural solutions in the context of Software Product Line (SPL) is proposed. Obviously, this approach is based on our AQOSA framework (described in Chapter 4). This novel search-based method produces a set of solutions which are suitable for the range of products defined by various feature combinations. In this SPL-aware method, AQOSA will also consider feature models as input to the framework and take into account the relationship between the software components in the architecture and features in the feature model. Hence, the approach applies the optimization techniques to each product of the SPL. After that, it analyses the commonality of the optimal solutions and proposes a set of solutions which are suitable for the range of products defined by various feature combinations.

This chapter is structured as follows. Section 8.1 discusses the optimization process for a software product line. It proposes a new process for the purpose of SPL-aware architecture optimization. It also discusses about required tasks for modelling and optimization. Then, Section 8.2 introduces an algorithm for obtaining common architectural solutions out of multiple Pareto fronts for multiple products within a product line. As validation of the approach, an experiment and its results are presented in Section 8.3. Finally, Section 8.4 summarizes this chapter.
8.1 Optimization for SPL

As discussed in Chapter 4, AQOSA is a framework which uses genetic algorithm (GA) optimization methods for automated software architecture design. The framework supports analysis and optimization of multiple quality attributes including response time, processor utilization, bus utilization, safety and cost. In order to extend the framework, for optimization in the context of product lines, the following extensions have been made:

- (i) the process of modelling has been upgraded to address the modelling of features and their relationship to their underlying components,
- (ii) the optimization process has been extended to optimize for a set of different products,
- (iii) the AQOSA intermediate representation (IR) model has been extended to be able to store feature model information,
- (iv) a new analysis step has been added after the optimization process, in order to determine commonality across the solutions for different products of the SPL among multiple resulted Pareto fronts.

In the following, Section 8.1.1 describes the general process for architecture optimization within the product line. The details of feature modelling and architecture modelling are discussed in Section 8.1.2. Section 8.1.3 discusses briefly how the optimization part works. Lastly, Section 8.2 describes the commonality analysis algorithm.

8.1.1 Process

Figure 8.1 depicts an overview of the process of our proposed method. It consists of the following steps which we can categorize into modeling, optimization and commonality analysis steps:

1. Feature modeling
2. Architecture modeling
3. Connecting features to their implementing software components
4. (Optionally) Defining target products in the product line
5. Evolutionary optimization of software architecture for every product
6. Commonality Analysis among the Pareto front solutions of all products
Figure 8.1: Process of finding similar optimal architectural solutions from Pareto fronts
**Step 1** and **Step 2** are for designing a feature model and an architecture model, respectively. In **Step 3**, we define the relationship between each feature in the feature model and the software components in the architecture model, to specify which components provide the required functionality of that feature.

In **Step 4** which is optional, the architect defines for which predefined products he would like to apply the optimization. These first four steps are described in Section 8.1.2.

**Step 5** which is the evolutionary optimization of software architecture will be repeated for every product selected in **Step 4**. The results of this step is $p$ sets of optimal solutions (Assume, the number of predefined products in **Step 4** is $p$). This step is discussed in Section 8.1.3.

The final step (**Step 6**) is the analysis of commonality between all of solutions in all Pareto fronts. The proposed approach for this analysis is presented in Section 8.2. The output of this step is a final set of optimal solutions that are applicable for a range of products in the SPL.

---

**Listing 8.1: Clafer model**

```plaintext
1 car
2  xor ignition_switch
3    key_ignition
4    button_ignition
5    interior_lights
6   xor dashboard
7     simple_dashboard
8     extended_dashboard
9   airbag_system ?
10  front_airbag
11  sides_airbag ?
12  passengers_airbag ?
13  antilock_braking_system ?
14  traction_control_system ?
15   xor stability_control_system ?
16     basic_skid_control
17     extended_skid_control
18   xor cruise_control ?
19     basic_cruise_control
20     adaptive_cruise_control
21    fullyAdaptive_cruise_control
22  theft_alarm ?
23  park_assist ?
24 [stability_control_system => traction_control_system]
25 [traction_control_system => antilock_braking_system]
26 [basic_cruise_control => basic_skid_control]
27 [adaptive_cruise_control => extended_skid_control]
28 [fullyAdaptive_cruise_control => extended_skid_control]
29 [button_ignition => !simple_dashboard]
```
8.1.2 Modelling

Feature modelling

In order to define the various features in the product line, the architect should model them in a product line modeling language. Clafer is one of the best-known feature modelling languages and it is used for our feature modelling step. Antkiewicz in [ABM+13] defines Clafer as a lightweight yet expressive language for structural modelling. It supports feature modelling and configuration, class and object modelling, and metamodeling.

Listing 8.1 shows an example of a Clafer model. It is the textual representation of the feature model of our case study in Figure 8.2. In Section 8.3, the details of this model have been described. As can be seen, there are some notations in this textual representation:

- `?`: Question mark notation means optional feature. For example, in Listing 8.1, `airbag_system` is an optional feature for a car. However, if a car has `airbag_system`, then it should have `front_airbag` as well.

- `[ ]`: Square brackets notation defines constraints in the feature model. These constraints can be used to define compatible or incompatible features. They are also used to define feature dependencies.

- `=>`: This notation defines feature dependencies. For example, in Listing 8.1, `stability_control_system` and `traction_control_system` are both optional features. However, if a car has `stability_control_system`, then it should have `traction_control_system` as well.

- `!`: Exclamation mark means logical negation. For example, in Listing 8.1, if a car has `button_ignition`, then it is not possible to have `simple_dashboard` at the same car.

Architecture modelling

The architecture modelling activities is similar to what is already discussed in Section 4.3. The only difference here is that the architect should model features in AQOSA IR model as well. We know from Chapter 4 that Features branch of AQOSA IR model in Figure 4.4 is generally optional. However, it is needed to be defined in optimization in SPL context. By using that branch (sub section) in AQOSA IR model, the architect would be able to define what feature in Clafer model is related to which component software architecture. This information will be stored by realize meta-reference which connects Feature meta-class to Component meta-class. It is not needed to model the relation between features themselves, the realization of each feature by its implementing components is the focus of this step.
Connecting features to their underlying software components

After defining the feature model (Step 1) and the architecture model (Step 2), in this step (Step 3) the architect needs to define the connections of features with their implementing components. These connections will also be stored as part of our AQOSA IR model. As Figure 4.4 shows, the reference between Feature entities and Component entities (which is called "realize") defines what are the underlying components for each specific feature. For the consistency between Clafer model and Eclipse model, we use the same feature identifier in both models.

Defining target products in the product line

This last step for the modelling is optional which enables the architect to define some of predefined products among all possibilities within product line. In this way, the architect can target for optimizing the exact products in the product line. These predefined products should be modelled in Clafer language. The tool validate the correctness of them and use them during the optimization process.

However, it is possible to skip this step. In this case, we use the Alloy Analyser [Depb] to explore the features space. Alloy Analyser is a solver that takes model constraints and finds structures that satisfy them. The architect only needs to configure the number of products in the product line which he would like to have. The tool by using the Alloy Analyser calculates all possible feature configurations. Then, it sorts them based on their resource density. And finally, it picks the evenly distributed products in a way that covers whole parts the product line.

8.1.3 Optimization

The optimization process for the SPL consists of exactly the same steps which are described in Section 4.4. But, this process should be executed for each product in the product line separately. For example, if $p$ is the number of defined products by the architect in Step 4, then the AQOSA optimizer should be applied $p$ times once for each product in the SPL. The result would be $p$ Pareto fronts.

8.2 Commonality Analysis

Because the architecture optimization process target optimization of multiple objectives at the same time, it produces solutions that vary from each other and it is very rare to have two identical solutions. Therefore, we need an algorithm which allows us to pick identical solutions, or at least, solutions with minimum distance from each other. We call it "Commonality Analysis Algorithm". Hence, for analysing the commonality of the solutions, we use a distance algorithm which is described in Section 8.2.1.
Commonality Analysis

Input: all Pareto fronts (frontSets)
Output: a list of architectural solutions

foreach Solution s1 in one Front from frontSets do
  foreach Front f in remaining Fronts do
    foreach Solution s2 in f do
      calculate the distance between s1 and s2;
      if distance < Δ then
        break and continue with next Front f;
      end
    end
    break and continue with next Solution s1;
  end
  end
  Store s1 in the list of common solutions;
end

Algorithm 1: Commonality analysis algorithm

The final goal is to find a set of optimal solutions which appeared in all Pareto fronts or that differ with some maximum distance (Distance is defined as the number of changes which are need for converting one architecture into another). We call this maximum distance as Δ. It is needed to configure Δ as a parameter of the method, before running the optimization process. The commonality analysis algorithm works as described in Algorithm 1 (Assume p is the number of products which we optimize the architecture for).

8.2.1 Solutions Distance Calculation

The core part of the commonality analysis algorithm is the calculation of distance between solutions. To do that, we use an algorithm inspired by the Levenshtein distance algorithm [PB99]. Our algorithm calculates the number of changes needed in a hardware platform, to change one solution into another. In other words, each solution contains a list of processors and a list of buses. Two solutions are considered equal when they do not require any hardware changes in order to change one solution into another. However, two solutions are not equal when they require some changes in order to match the hardware of the other solution.

A change can be one of the following: substitution, addition or removal of a hardware node; a processor or a bus. The costs of these changes are not the same because changing a processor with another is always easier than modifying (add/remove) an existing topology, which can also lead to modification of the buses and their connec-
Input: Two Solutions (s1 and s2)  
Output: an integer number as their distance

store two solutions as list of resources;  
find which list is longer (longer and shorter);  
foreach Resource r in longer list do
  if shorter list contains r then
    remove r from both list;
  end
end
substitution changes = the remaining number of resources in shorter list;  
addition changes = (the remaining number of resources in longer list) - substitution changes;  
distance = (substitution changes) + (ω * addition changes);

Algorithm 2: Solutions distance calculation algorithm

8.3 Experiment for SPL-aware Optimization

To explore the optimization problem addressed in this chapter and to evaluate our proposed method, we extended a case study based on an existing sub-system from the automotive industry (described in Section 5.3). We added new features like cruise control system, airbag system, park assist system, etc., to increase the complexity of the optimization problem. The goal of the case study is to find a set of optimal architectural solutions that are applicable, with a minimum of changes, to all of our defined products within the product line with a minimum of changes. This is a relevant problem in the automotive industry, because of a large number of vehicle models with varying feature content that must be supported by the software architecture [EG13].

Figure 8.2 depicts the feature model of the case study (Listing 8.1 shows its textual representation). A car in our model should consist of at least an ignition switch, interior lights, and dashboard. In addition, it may optionally have the following systems: airbag system, anti-lock braking system, traction control system, stability control system, cruise control system, theft alarm system and park assist system. The ignition switch can be either key ignition or button ignition technology. Likewise,
Figure 8.2: Case study feature model
the dashboard can be either a simple dashboard system or an extended dashboard system and also the stability control system can be either basic or extended. For the cruise control system, there are three possibilities: basic cruise control system, adaptive cruise control system and fully adaptive cruise control systems. For cars with an airbag system, the front airbags are compulsory. However, it can optionally have side airbags and/or passengers airbags. Green arrows in the figure demonstrate the dependencies between the features. In addition, red arrows show mutually exclusive features in our feature model (e.g. button ignition is not compatible with simple dashboard feature).

### 8.3.1 Products of SPL

By exploring the feature model, we figured out that, in total 480 various combinations of feature sets are possible. However in this experiment, we defined the following five imaginary cars as our products in the product line. They start from a very low-end model to the high-end full feature product. We have defined them in a way that the combination of features in each product are logical and feasible in real-world models. More importantly, they cover the extremes of the design space. Figure 8.3 shows the distribution of CPU resource claims among all 480 possible products.

To make sure that we cover the whole product line, we calculated the amount of processing resources that the underlying components claim for each feature configuration. Then, we sorted the list of configurations based on that measure. The result shows that our defined products are positioned in rank 5, 104, 264, 389, and 480 of the sorted list, respectively. Hence, we have covered important parts of the product line spectrum with these five products.

![Resource Claims of Feature Configurations (by cpu usage)](image)

*Figure 8.3: Resource claims (CPU cycles) for all possible products in the feature model*
Car1

Figure 8.4 shows the feature configuration of the first car. Car1 is a low-end feature product. It only consists of front airbags as optional feature, the rest are all mandatory features that the car should support. Since it is a low-end product, it is designed with a simple dashboard which means that it can not support button ignition and therefore it comes with the key ignition feature.

Figure 8.4: Feature configuration for Car1
Car2

Figure 8.5 shows the feature configuration of Car2. It has all features of Car1, plus the basic skid control functionality. And because of that feature, it has to have traction control and anti-lock breaking systems.

Figure 8.5: Feature configuration for Car2
Car3

Car3, as depicted in Figure 8.6, has an extended dashboard with key ignition. It has a basic cruise control system and therefore it should have basic stability control and traction control and anti-lock breaking systems.

Figure 8.6: Feature configuration for Car3
Car4

Figure 8.7 depicts the feature configuration of Car4. In addition to the features of Car3, it has a button ignition and an adaptive cruise control system. It also provides the theft alarm functionality.
Car5 (Figure 8.8) is the very high-end product in our product line. It has all the features (with the best one selected in mutually exclusive cases).

Figure 8.8: Feature configuration for Car5
8.3.2 Experiment Setup

To define the experiment context, first we need to determine which quality attributes are our objectives for optimization. So, we set these five quality attributes as our optimization objectives: bus utilization, cost, CPU utilization, response time, and safety.

Second, we set the hardware repository. The following repository of hardware gathered based on data from the industrial case study:

- 10 Processors: ranging over 5 various processing speeds from 10 MIPS to 100 MIPS. Each of these has two levels of failure rates. A processor is more expensive if it has more processing power or a lower failure rate.

- 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.

Finally, we run AQOSA 60 times using the NSGA-II algorithm with these adopted parameter settings: initial population size ($\alpha$)=2000, parent population size ($\mu$)=100, number of offspring ($\lambda$)=50, archive size=25, number of generations=500, and all quality attributes were aimed to be minimized. For commonality analysis, we configured $\omega$ (the ratio for additional change distance to substitution change distance) to 3. Moreover, we set $\Delta = \{2, 3, 4, 5, 6, 7\}$ and compared the results of the algorithm for changing the $\Delta$ parameter.

\[
\begin{align*}
\text{Node1: 100MHz, €15} & \quad \text{Node2: 600MHz, €70} \\
\text{Bus1: 512 kbps, 2ms delay, €1} & \quad \text{Bus2: 128 kbps, 8ms delay, €0.5} \\
\text{Node3: 100MHz, €15} & \\
\end{align*}
\]

Figure 8.9: Proposed common solution which is optimal for Car1
Experiment for SPL-aware Optimization

Node1: 100MHz, €15
Node2: 400MHz, €50
Node3: 1000MHz, €120
Bus1: 512 kbps, 2ms delay, €1
Bus2: 512 kbps, 2ms delay, €1

(a) Similar solution which is optimal for Car2

Node1: 600MHz, €70
Node2: 100MHz, €15
Node3: 800MHz, €100
Bus1: 512 kbps, 2ms delay, €1
Bus2: 16 kbps, 50ms delay, €0.1

(b) Similar solution which is optimal for Car3

Node1: 1000MHz, €120
Node2: 100MHz, €15
Node3: 100MHz, €15
Bus1: 512 kbps, 2ms delay, €1
Bus2: 128 kbps, 8ms delay, €0.5
Node4: 600MHz, €70

(c) Similar solution which is optimal for Car4

Figure 8.10: Similar solutions in other Pareto fronts

Node1: 600MHz, €70
Node2: 400MHz, €50
Node3: 1000MHz, €120
Node4: 600MHz, €70
Node2: 100MHz, €15
Node3: 100MHz, €15
Bus1: 512 kbps, 2ms delay, €1
Bus2: 128 kbps, 8ms delay, €0.5

(d) Similar solution which is optimal for Car5
### 8.3.3 Experiment Results

In this section, we discuss the results of the aforementioned experiment with 60 runs. Table 8.1 shows the average number of found common solutions over 60 runs among various Pareto fronts. As it shows, by increasing the $\Delta$ parameter, our algorithm would be able to find more common solutions. Figure 8.11 demonstrates the boxplot of the data that Table 8.1 shows.

Figure 8.9 and Figure 8.10 represent one of the proposed solutions, from one arbitrary run. In each run, five Pareto fronts were generated for all the five products. Each front contained 25 architectural solutions, making a total search space of 125 solutions for commonality analysis. So, it explored five Pareto fronts to find common solutions across all five configurations.

Figure 8.9 shows an optimal solution for Car1 that has acceptable (within $\Delta$ range) distance from solutions in other Pareto fronts. This means that these common solutions require only few substitution and/or additional changes in order to become a solution for other products. For example, Figure 8.10a shows the optimal solution for Car2 which has distance of 3 with the solution represented in Figure 8.9. And similarly, Figure 8.10b depicts the similar solution for Car3, Figure 8.10c optimal solution for Car4, and Figure 8.10d optimal solution for Car5.

### 8.3.4 Validation

As mentioned earlier, the goal of this case study was to find a set of optimal solutions that are applicable, with a minimum of changes, to all of our defined products within the product line. This is a relevant problem in the automotive industry, because of a large number of vehicle models with varying feature content that must be supported by the software architecture. The feature content of a specific car can be decided by the customer to a high degree. On the other hand, due to fierce competition within the automotive industry, development cost needs to be decreased by reusing as much software as possible. Therefore, the problem studied in the case study is a real problem in the automotive industry.

We applied our proposed method to an extended version of a case study based on an existing sub-system from the automotive industry. In the original version of the case study, we applied AQOSA to the same requirement specifications used by the architects at the automotive company when designing the current realization. In this

<table>
<thead>
<tr>
<th># Solutions</th>
<th>$\Delta = 2$</th>
<th>$\Delta = 3$</th>
<th>$\Delta = 4$</th>
<th>$\Delta = 5$</th>
<th>$\Delta = 6$</th>
<th>$\Delta = 7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>0.88</td>
<td>1.52</td>
<td>2.13</td>
<td>2.90</td>
<td>4.00</td>
<td>5.15</td>
</tr>
</tbody>
</table>

Table 8.1: Average number of common solutions over 60 runs among various Pareto fronts
chapter, we added new features to the case study like cruise control system, airbag system, park assist system, etc., to increase the complexity of the optimization problem. By doing this we ended up investigating a rather complex sub-system of realistic scale. Thus, the case study is partly based on real requirement specifications specifying a realistic sub-system that is relevant in an automotive context, and at the same time limited enough to explore and demonstrate how our search-based approach supports in solving this optimization problem. The resulting set of architecture solutions proposed by our method contains good candidates for manual assessment in later phases of the architecture design process for a product line. This is confirmed by comparing the set of architecture solutions to the existing sub-system architecture. However, the accuracy and relevance of the proposed architecture solutions are not the primary goals of this exploratory case study. Moreover, one of the advantages is that our approach can be executed as a distributed search approach which increases the speed of reaching to the results.

Regarding threats to the validity of our results, the main type is external validity which concerns generalization of the results outside the context of the study [RHRR12]. The case study was conducted based on an existing sub-system from one automotive company using specifications, software, and data from that particular company. The results of the case study suggest that the architecture optimization framework can be applied to other software product lines for embedded systems, but this needs to be assessed by conducting additional case studies in other contexts.

![Boxplot of the number of found common solutions for various \( \Delta \)](image)

**Figure 8.11:** Boxplot of the number of found common solutions for various \( \Delta \)
8.3.5 Interpretation of results

The objective of the case study is to answer the research question introduced in Section 1.2: “In what aspects can search-based approaches improve the process of designing a set of software architectures with a minimum of manufacturing changes for a range of products in a software product line?”. This section will provide answers to the research question.

The purpose of the AQOSA framework is to support the architect in early phases of architecture design, and especially with searching large solution spaces while considering various quality constraints. So, the underlying approach is to combine the domain knowledge and experience of the architect with the optimization, simulation, and analysis skills of the AQOSA framework. The case study illustrates how this combination can solve a practical problem. The domain knowledge and experience of the architect is needed when defining the problem to be solved, when creating the models used as input to AQOSA and when evaluating the output from AQOSA. The optimization, simulation, and analysis skills of AQOSA are needed when searching a large theoretical design space, when analysing a large number of potential solutions, and when considering multiple quality attributes.

The case study in this chapter confirms that it is important to consider all attributes and constraints when designing the architecture. If only a subset of the attributes are considered during design, there is a risk to select a solution that is infeasible with respect to other equally important attributes. The challenge then becomes to search a large design space while meeting increasing time-to-market demands. This challenge is even increased in the context of software product lines, since architectural solutions must be found for several products in the product line. This challenge was explored in the case study using the AQOSA framework which required a manual effort of 5 man days in total, and around 250 minutes of execution time on a powerful computer with a 12-core (each core runs 2.67GHz and 12MB cache) processor and 48GB memory. The manual effort was needed for modelling activities and for analysing the results.

8.4 Summary

This chapter proposed a novel search-based method for finding optimal software architectural solutions which are applicable for a range of products in a product line. By introducing this approach, the AQOSA framework is extended to support multiple products at the same time. Earlier chapters discussed the AQOSA framework in a way that it aims at supporting architects in finding optimal architecture solutions in complex design situations with many potentially conflicting quality attributes. This chapter reported that AQOSA can be employed by architects in the context of software product lines as well.
We demonstrated the application of our proposed approach on an exploratory case study based on an existing sub-system from the automotive industry. The case study showed that while optimization techniques can find efficient solutions regarding all quality attributes, our method identifies similar optimal solutions that are applicable to the range of products in the software product line.

To summarize, we showed that our search-based approach can improve the process of designing a set of software architectures for a range of products in a software product line in the following aspects:

- modelling the relationship between feature model and component model,
- evolving architectural solutions for the range of products in the SPL at the same time,
- find similar optimal architecture solutions that are applicable to the range of products in the SPL.