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
Chapter 4

AQOSA Framework

To be able to investigate the research questions of this dissertation, a new meta-heuristic optimization framework for automated software architecture design has been developed based on the previous work and knowledge (e.g. [BCdW06], [LCL10]). The tool has been developed with new implementation from scratch. Our tool is named AQOSA (Automated Quality-driven Optimization of Software Architectures). This chapter discusses the details of this framework. The tool is open source and its source codes are accessible via http://bitbucket.org/retemaadi/aqosa.

The AQOSA framework has aimed the following goals to enable us to answer the research questions described in Section 1.2:

1. Be able to support multiple quality attributes (a must have), and also gives the possibility to extend quality attributes with external evaluators (a could have).

2. Increase the software architecture exploration space by enabling support for multiple degrees of freedom in varying architectural solutions (a should have).

3. Be independent from architectural modelling languages. So, it would be able to interoperate with various architectural modelling languages (e.g. AADL [FGH06], UML/MARTE [OMGb], PCM [BKR09], ROBOCOP [Inf03]) (a must have).

This chapter is structured as follows. First, Section 4.1 gives an overview about the AQOSA framework. Section 4.2 details the structure and the composition of the AQOSA tooling modules which are the next four sections. Hence, the modelling part, the optimization part, the solution part and the evaluation part are described in Sections 4.3, 4.4, 4.5, 4.6 respectively. Section 4.7 demonstrates two possibilities for monitoring Pareto fronts within the optimization process. After that, the complexity of the used algorithm in our approach is discussed in Section 4.8. And finally, Section 4.9 summarizes this chapter.
4.1 Framework Overall Process

AQOSA is a framework which uses a Genetic Algorithm (GA) optimization approach for automated software architecture design. The framework supports analysis and optimization of multiple quality attributes of the system including response time, processor utilization, bus utilization, safety and cost. Figure 4.1 shows the architecture of the AQOSA framework. It uses an architectural Intermediate Representation (IR) model for describing the architectural design. It takes the following as input:

- An initial functional part of the system (i.e. components that provide the needed functionality and their interactions with other components),
- A set of typical usage scenarios (includes triggers to create workloads),
- A list of objective functions (determines which architecture properties should be optimized),
- A repository of components which contains all specifications related to hardware and software components instances.

Then, AQOSA iterates through the following steps:

1. Generate a new set of candidate architecture solutions: To this end, 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 candidates.
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 invokes up the property-specific analysers to evaluate these models.

3. Select a set of (so far) Pareto-optimal solutions.

4. Iterate to step 1 until some stopping criterion holds. This can be a maximum number of generations or a criterion on the objective functions.

### 4.2 AQOSA Tooling Design

The AQOSA tool, developed in Leiden University as part of this dissertation, helps software architects to find optimal solutions for component-based software systems in the early stages of software architecture design. Figure 4.2 depicts the decomposition of the tool. It consists of the following major parts:

1. **Modelling** part: It includes two modules (Scenario and Repository modules), which are regarded as input for the tool. They are described in Section 4.3.

![Figure 4.2: AQOSA tooling parts](image-url)
2. **Optimization** part: Using evolutionary algorithms, this part tries to explore the design space for finding optimal solutions. It employs the Solution module to vary architectural solutions and the Evaluator module to measure them. More details are given in Section 4.4.

3. **Solution** part: Using a state-of-the-art genotype for software architecture it can generate architectural solutions in a broad range of degrees of freedom. More details are discussed in Section 4.5.

4. **Evaluation** part: The responsibility of this part is analysing different quality attributes for the candidate solutions. It uses external evaluators for this purpose. For example, it uses the queuing network analysis to evaluate performance attribute and the failure analysis to evaluate safety attribute. This part is described in Section 4.6.

### 4.3 Modelling Part

Because AQOSA is designed to optimize architectures in a wide range of domains, it aims to be independent from specific modelling languages. Hence, it uses its own internal architecture representation, AQOSA intermediate representation (AQOSA IR). AQOSA architecture modelling is defined by means of the Eclipse EMF [Ecl]. The AQOSA IR model integrates multiple quality modelling perspectives (such as performance, safety, etc.).

Figure 4.4 represents a simplified view of the AQOSA IR meta-model. It consists of four major parts: Assembly, Scenarios, Repository and Objectives.

- **Assembly**: This part includes software components and their assembly for delivering system functionalities. Every component provides some services, and interaction between different components are defined by flows and actions within flows.

- **Scenarios**: This part defines expected scenarios for the system. Thereby, the architect can define best-case, worse-case or normal-case for the system. It stores real-time constraints of the system such as expected completion time and deadline.

- **Repository**: This part stores various possible choices for software and hardware components: such as processors, buses and component implementations. Based on this repository, the AQOSA framework is able to change the topology of the candidate solution, or assigned processor for each node, or assigned bus instance for each bus line, etc. This part contains required specifications of each possible hardware or software.
Objectives: This part defines the objectives which the framework optimizes.

Figure 4.3 demonstrates a screenshot of the tool while modelling an architecture optimization problem. It shows the tree structure of the AQOSA model, while the architect is defining the scenarios and assigning the properties for various components as well. A sample of an AQOSA IR model (consists of its details objects and attributes) is presented in Appendix A. The sample is a model of the case study system that is described in Section 5.3.

4.4 Optimization Part

In general, design of software architecture has to address multiple contradicting quality attributes. Various global optimization techniques have been used in handling complex engineering problems. Younis et al. [YD10] compared several optimization methods and revealed the pros and cons of these global optimization methods. They classified Global Optimization (GO) methods into two main categories: deterministic methods and stochastic methods.

The problem of optimizing software architecture is a non-linear and discontinuous problem, i.e. small changes in the architecture design can have a very large impact.
Figure 4.4: AQOSA Intermediate Representation (IR) simplified meta-model
on the different quality attributes. So, the search space is combinatorial and discrete. Moreover, a large number of alternative designs exists in the search space. In this context, deterministic approaches do not perform well. Instead, stochastic methods are a better fit for solving this kind of problems. Within stochastic methods, Younis describes the following strengths and weaknesses [YD10]: Simulated Annealing (SA) algorithms emulate the annealing process on how liquid freezes or metal re-crystallizes in cooling. Simulated annealing is easy to implement, although the method converges slowly and it is difficult to find an appropriate stopping rule. Therefore, because of the convergence speed the SA method is not an option for our method. Particle Swarm Optimization (PSO) has also been applied to solve practical optimization problems and proved to be one of the promising and successful methods. PSO shares many similarities with evolutionary computation techniques, such as Genetic Algorithms (GA). Genetic algorithms (GAs) are a class of search procedures based on the mechanics of natural genetics and natural selection. However, PSO design paradigm is mainly suited for continuous vector spaces and not for combinatorial optimization. Moreover, unlike GAs, PSO has no evolution operators such as crossover and mutation. Because of the need for using intelligent operators as one of the key features in AQOSA framework, PSO is not an option for our method, as well. Hence, Genetic Algorithm (GA) is chosen as optimization method for the AQOSA framework.

In the following, first evolutionary algorithms supported by the AQOSA framework are mentioned. After that, the implemented degrees of freedom for our architecture optimizer are discussed.

4.4.1 Evolutionary Algorithms

Due to the conflicts of quality attributes in the software architecture design, AQOSA uses Evolutionary Multi-Objective Algorithms (EMOA) to evolve the architecture. It has been implemented based on the Opt4J optimization framework [LGRT11][Depa]. The system designer can choose one of the following GA algorithms for his design problem:

- **NSGA-II** (non-dominated sorting based multi-objective evolutionary algorithm): It is one of the most widely used EMOA techniques and has been proposed by Deb [DAPM02]. It has a selection operator which uses non-dominated sorting, and crowding distance. The non-dominated sorting makes sure that the points converge to the Pareto front. And the crowding distance sorting makes the points spread out across the Pareto front.

- **SPEA2** (an improved version of Strength Pareto Evolutionary Algorithm): It has been suggested by Zitzler and Thiele [ZLT02], and it is also widely used. It uses alternative ways for convergence and diversity compared to NSGA-II.
• SMS-EMOA (S-Metric Selection Evolutionary Multi-Objective Algorithm): It has been proposed by Emmerich, Beume and Naujoks [EBN05] [BNE07]. It is a representative of the class of hypervolume-based EMOA, which recently gained popularity in the EMOA field.

A comparison of EMOA algorithms for our specific domain, software architecture domain, is discussed in Section 5.1.

4.4.2 Degrees of Freedom

When an architectural design is created, usually, there are still variation to the solution 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 and wrappers. However, the optimizer should consider only the variations of architectural designs which do not modify the interfaces used in the architecture in order to guarantee that the optimization process does not change the functionality of the system.

In the following, the degrees of freedom which are implemented by the AQOSA framework are listed:

**Number of hardware nodes**

If an architectural model contains $n$ software components, then these can be deployed on a number of hardware nodes, ranging between a minimum of 1 and a maximum of $K\cdot n$ hardware nodes (for some natural number $K > 0$), because the number of nodes in an architecture is finite. Adding more hardware nodes may provide more processing capacity and therefore may yield better performance. On the other hand, removing hardware nodes may reduce the total cost of the system.

**Number of connections between hardware nodes**

If $n$ hardware nodes have been chosen for the deployment of components, then the maximum number of possible connections between hardware nodes can be calculated by: $\text{Max}_c(n) = \frac{n(n-1)}{2}$.

This maximum represents the case that all of the nodes are connected 1-by-1 together by a dedicated communication line. It is also possible to assume redundant connections between nodes or more interconnections between the connections themselves. In these cases this number could be even higher. But we assume $\text{Max}_c(n)$ as the maximum number of connections because redundancy of connections is rare in architecture design. As a minimum, it is possible to consider a single central bus which connects all of the nodes. This DoF has significant impact on performance and cost of the system.
Network topology

By definition, network topology is the layout of interconnections between hardware nodes. For example, Figure 4.5 shows some possible topologies for connecting a network with 4 nodes. In other words, even with the same number of nodes and the same number of connections, network topologies might be different. Even with the same number of nodes and connections, different topologies can still represent different architectures and the number of possibilities can be very large, which its number is in $O(2^{n^2})$. The impact and importance of this DoF is discussed in Chapter 6. It is important to consider that not all possible topologies are valid and therefore AQOSA performs a validation process before doing evaluation. This process is described in Section 4.5.2.

![Figure 4.5: Three possible topologies for a 4-node network](image)

Software on hardware allocation

Given a hardware network topology, the allocation of software components on hardware nodes is another degree of freedom. This degree of freedom defines which software component executes on which hardware component. It also has large effect on the processors’ utilization and system’s performance.

Software components selection

Different components (e.g. developed by different vendors) that implement the same functionality are considered as different architectural alternatives. Our assumption is that one component can replace another, if and only if they implement the same functionality. This assumption prevents the solution from the violation of the system’s functionality.

Hardware components selection

This DoF entails that each hardware node can be replaced by another hardware node in the repository. These nodes may be different in processing speed, energy consumption, failure probability and cost. Hence, it has large effect on all quality aspects.
Figure 4.6: Optimizer module class diagram
Communication lines selection

This degree of freedom is similar to the hardware components selection, but is aimed at selection of communication lines. They might be different in bandwidth, communication delay, failure probability, as well as cost.

4.4.3 Optimizer Module Implementation

Figure 4.6 depicts the class diagram of the Optimizer module and its relation with the Solution module. ArchProblem is the main class which reads an AQOSA IR model and optimization parameters as well. Then, it generates an initial population by calling the ArchCreator class. The ArchCreator class creates the ArchGenotype class which consists of four genomes. The ArchDecoder class decodes or translates an ArchGenotype into an ArchSolution which is also described in Section 4.5.3. The ultimate aim is to evolve genes to find optimal solutions, therefore, the ArchProblem class repeatedly calls the ArchEvaluator to analyse quality attributes for each solution. With the help of evolutionary algorithms (described in Section 4.4.1), the AQOSA framework keeps candidate optimal architectures and discards the others.

4.5 Solution Part

This section explains how the AQOSA framework encodes an architectural candidate solution for the optimization process. Because AQOSA uses a genetic algorithm for its optimization part, the architectural solutions need to be encoded in genetic form.

4.5.1 Architecture Genotype Structure

Table 4.1 shows the structure of the genotype which is used in the AQOSA framework. This genotype consists of a set of four genomes: (1) the Deployment genome, (2) the Nodes genome, (3) the Communication genome, and (4) the Connection genome.

1. The deployment genome (Table 4.1a) shows for each software component which implementation is deployed on which hardware node. It encodes the ‘software components selection’, ‘software on hardware allocation’, and the ‘number of hardware nodes’ DoFs.

2. The nodes genome (Table 4.1b) represents which hardware variant is selected for each node in the system. It encodes ‘hardware components selection’ DoF. The specification for each hardware node includes processing clock rate, range of failure probability, and cost.
3. The communication genome (Table 4.1c), like the nodes genome, represents which hardware variant is selected for each communication line. It encodes the ‘number of connections’ and the ‘communication lines selection’ DoFs. Their specification includes bandwidth, transmit delay, and cost.

**Table 4.1: AQOSA framework genotype**

<table>
<thead>
<tr>
<th>Component$_1$</th>
<th>Component$_2$</th>
<th>Component$_3$</th>
<th>...</th>
<th>Component$_c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implmnt. $&lt; i_1 &gt;$ deployed on Node $n_{i_1}$</td>
<td>Implmnt. $&lt; i_2 &gt;$ deployed on Node $n_{i_2}$</td>
<td>Implmnt. $&lt; i_3 &gt;$ deployed on Node $n_{i_3}$</td>
<td>...</td>
<td>Implmnt. $&lt; i_c &gt;$ deployed on Node $n_{i_c}$</td>
</tr>
</tbody>
</table>

(a) **Deploy Genome**

<table>
<thead>
<tr>
<th>n (No. of Nodes)</th>
<th>Node$_1$</th>
<th>Node$_2$</th>
<th>Node$_3$</th>
<th>...</th>
<th>Node$_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>HW Spec. $&lt; h_1 &gt;$</td>
<td>HW Spec. $&lt; h_2 &gt;$</td>
<td>HW Spec. $&lt; h_3 &gt;$</td>
<td>...</td>
<td>HW Spec. $&lt; h_n &gt;$</td>
<td></td>
</tr>
</tbody>
</table>

(b) **Nodes Genome**

<table>
<thead>
<tr>
<th>l (No. of Lines)</th>
<th>Line$_1$</th>
<th>Line$_2$</th>
<th>Line$_3$</th>
<th>...</th>
<th>Line$_l$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus Spec. $&lt; b_1 &gt;$</td>
<td>Bus Spec. $&lt; b_2 &gt;$</td>
<td>Bus Spec. $&lt; b_3 &gt;$</td>
<td>...</td>
<td>Bus Spec. $&lt; b_l &gt;$</td>
<td></td>
</tr>
</tbody>
</table>

(c) **Communication Lines Genome**

<table>
<thead>
<tr>
<th>Node$_1$</th>
<th>Node$_2$</th>
<th>Node$_3$</th>
<th>...</th>
<th>Node$_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>True/False</td>
<td>True/False</td>
<td>True/False</td>
<td>...</td>
<td>True/False</td>
</tr>
</tbody>
</table>

Buses-to-Nodes Connection Matrix

<table>
<thead>
<tr>
<th>Line$_1$</th>
<th>Line$_2$</th>
<th>Line$_3$</th>
<th>...</th>
<th>Line$_l$</th>
</tr>
</thead>
<tbody>
<tr>
<td>True/False</td>
<td>True/False</td>
<td>True/False</td>
<td>...</td>
<td>True/False</td>
</tr>
</tbody>
</table>

(d) **Connections Genome**
4. The *connection genome* (Table 4.1d) is the part of the genotype that represent architectural topology by the means of two Boolean matrices. These matrices list the connections between buses and nodes, and amongst buses themselves. Each cell in the matrix can be True or False, where True means this particular bus and particular node (or that particular bus) are connected; and False means those are not connected. They encode the ‘network topology’ DoF.

**4.5.2 Genotype Validation**

Evolutionary algorithms in AQOSA can apply various genetic operators such as Copy, Mutate or Crossover to the genotype for generating new architectural solutions as offspring. However, the validity of these solutions is not guaranteed. Therefore, the optimizer performs sanity checks for each genotype in order to validate it. In this process, those solutions which can not satisfy the functionality of the system will be omitted from being sent to the evaluation process. For example, if the model defines that component3 should communicate with component5 and then the generated offspring deploys component3 on node2 and component5 on node4, this process should check whether there are any communication paths between node2 and node4.

**4.5.3 Solution Module Implementation**

Figure 4.7 depicts how the Solution module implements the architecture genotype. ArchGenome is an interface which all the architectural genomes should implement. As can be seen in the figure, DeployGenotype, NodesGenotype, EdgesGenotype, and ConnectionGenotype implement that interface. They represent the deploy genome (4.1a), the nodes genome (4.1b), the communication genome (4.1c), and the connection genome (4.1d) respectively.

**4.6 Evaluation Part**

The AQOSA evaluation part takes as input an evaluation model (e.g. queuing network or fault tree). An evaluation model is a transformation of the AQOSA IR and a candidate genotype for a particular quality attribute (e.g. response time or safety). The AQOSA framework feeds evaluation models to each evaluator and returns the results to the optimization part.

The AQOSA framework uses analysis tools for particular quality attributes as plugins. It is assumed that these analysis tools are developed and validated by domain experts. However, to examine the accuracy of AQOSA evaluation implementation, the results generated by AQOSA evaluation have been compared with results reported in relevant publications. In case of performance attributes, the QN implementation
Figure 4.7: Solution module class diagram
can achieve the same results as the results published in [WTVL06]. In case of FTA, the results have been compared with the results published in [FS10].

4.6.1 Evaluation Model Transformation

To explain how the transformation of an architectural solution to an evaluation model works, in the following two examples are given: (1) for queuing network transformation, (2) for fault tree transformation. For the purpose of the example, a very simple scenario is assumed which is depicted in Figure 4.8. The figure shows a software architecture that consists of two components: Component A and Component B. The start trigger calls Service X from Component A through Port P. Then, Service X passes data through Port Q to Service Y from Component B which receives the data through Port R. Subsequently, Service Y generates the required data as the output of the scenario.
In addition to this scenario, for evaluating an architecture based on a queuing network it is needed to know the hardware topology and the deployment of the candidate solution as well. Assume, the solution is a simple architecture, which contains two processing nodes and one bus which connects them together. Table 4.9 represent that architecture based on the AQOSA genotype structure. **Component A** has been deployed on Node 1 and **Component B** on Node 2. In this case, **Bus1** represents the connection between **Port Q** and **Port R**.

**Queuing Network Transformation**

To generate a Queuing Network (QN), the AQOSA evaluation part will create a QN as shown in Figure 4.10. Each processing node or communication line represents one resource. Hence, each is mapped to its own queue. In this network, **CPU1** can be taken by **Service X** (because **Component A** has been deployed on Node 1), **CPU2** can be taken by **Service Y** and **Bus1** can be busy when **Service X** passes the data to **Service Y**.

**Figure 4.10: Generated queuing network for sample scenario**

**Figure 4.11: Generated fault tree for sample scenario**
Because the AQOSA QN analysis is based on JINQS [Fie10] library, it is purely Java-based [Sun] implementation. Analysis of QN statistics after the simulation will provide required data for response time, CPU utilization and bus utilization objectives.

**Fault Tree Transformation**

To generate a fault tree, the AQOSA evaluation part will create a fault tree for safety analysis as shown in Figure 4.11. This figure represents that the system output failure depends on Service Y failure. Furthermore, Service Y failure itself depends on two nodes: failure of the signal on Port R, and CPU2 hardware failure. And repeatedly, the tree can be parsed to the bottom. By running a Monte-Carlo simulation, the AQOSA evaluator analyses the safety objective for the candidate solution.

**4.6.2 Quality Attributes**

In the following, the quality attributes supported in the AQOSA framework are described.

**Response Time**

Response time refers to a time interval during which the response to an event must be executed. The time interval defines a response window given by a starting time and an ending time. These can either be specified as absolute times (time of day, for example) or offsets from an event which occurred at some specified time [BKLW95]. The ending time is also known as a deadline. AQOSA measures response time for each event in the system as offsets from the time at which the event happened, and then scores all events in the scenario based on predefined deadlines.

**Processor Utilization**

Processor utilization is the percentage of time during which a resource is busy. AQOSA measures this percentage for each processor in the architecture individually. AQOSA can be configured to return either average, minimum, or maximum processors utilization for a candidate architecture.

**Communication Line Utilization**

Like processor utilization, communication line utilization is the percentage of bandwidth that a communication line (or a bus) uses. Similar to above, AQOSA measures this metric for each bus in the architecture individually. Again, this metric can be configured to return either average, minimum, or maximum bus utilization.
It could be argued that utilization (either processor or communication line) is more an internal metric than a quality attribute. However, utilization can be considered as an indicator for other architecture quality attributes, such as extensibility. Because by choosing an architecture which reserves some free resources, the architect would be able to extend the system with more load in the future. Hence, using utilization in this way as a direct metric is more reasonable, first because the response time is calculated anyway, and second because measuring an indirect metric like extensibility is more challenging.

**System Safety**

Forster [FT09] claims: "Software does not fail randomly but will invariably fail again in the same way under the same conditions. While for mass-produced hardware parts it is possible to assign a failure probability, for software a similar assumption does not seem entirely realistic". Accepting this hypothesis, AQOSA assumes for each component, the output fails if either the input fails or the hosting hardware crashes. So, AQOSA analyses the corresponding fault tree for each system output based on these assumptions. Using Monte-Carlo sampling, AQOSA calculates the failure probability of each output based on its fault tree dependability on various inputs and also related hardware nodes probability of failures.

**System Cost**

The cost quality attribute is important from a market point of view. Fortunately, it is easily calculated by adding the cost of used software components, hardware nodes, and communication lines.

**Extension of Quality Attributes**

The architecture of the AQOSA framework facilitates the extension of AQOSA with new quality attributes. To this purpose, the following steps should be followed:

- An evaluator for that new quality attribute should be provided,
- An implementation for the transformation of an architectural solution to the proper evaluation model should be provided.

This evaluation model must be compatible with the input for the evaluator (provided in the previous step). Hence, the transformer component should transform a combination of the AQOSA IR and a candidate architecture model to the particular quality attribute evaluation model.

For example, it is easily possible to extend AQOSA for **power consumption quality attribute**. To this end, it is needed to implement a transformation component which:
Figure 4.12: Evaluator module class diagram
1. captures the processor types from AQOSA IR,
2. reads processor utilization values from queuing network simulation results, and transform them into an evaluation model.

The power consumption evaluator should be able to calculate the power consumption of that particular architectural solution based on those information.

4.6.3 Evaluator Module Implementation

Figure 4.12 depicts the class diagram of the **Evaluator** module. EvaluationThread is an abstract class which is the core of the module. Extending this class enables us to add evaluators for different quality properties. As can be seen in the figure, PerformanceEvaluator, SafetyEvaluator and CostEvaluator inherit from this class. Each quality evaluator is linked to a quality-specific analysis method.

4.7 Pareto front Monitoring

AQOSA tooling is implemented in a way which allows the architect to monitor the process of optimization. To this end, AQOSA offers both an application-based interface and a web-based interface. Hence, the architect is free to choose either execution of the optimization process on a local machine or execution in the cloud.

![AQOSA live Pareto front monitoring screenshot](image-url)
Application-based Interface

Figure 4.13 depicts a screenshot of AQOSA while Pareto front monitoring from the Java-based [Sun] application interface. Because it is developed based on the Opt4J framework [Depa], it uses the same visualization as the Opt4J framework. As can be observed, by using this interface it is possible to monitor multiple Pareto fronts and also convergence plots at the same time. In the progress bar it shows the progress of the genetic algorithm in terms of number of generations.

For example, in Figure 4.13, the architect is looking at live Pareto front of cost vs. CPU utilization. At the bottom of the window, he is monitoring two convergence plots: safety and bus utilization.

Web-based Interface

In Figure 4.14 a screenshot of AQOSA’s web-based interface is depicted. For using the web-based interface, it is required to first upload an AQOSA IR model to the cloud. After that, the architect can configure the optimization settings and execute the optimization process in the cloud. As can be seen in Figure 4.14, it is only possible to monitor one Pareto front at a time. Live web-based Pareto front charts have been implemented on top of using Google Chart [Goob].

For example, in the following screenshot, AQOSA’s web-based interface is deployed on the CloudBees cloud platform [Clo]. AQOSA’s web-based has been designed and implemented, so that it is also compatible with the Google App Engine platform [Gooa].

Figure 4.14: AQOSA web-based interface screenshot
4.8 Framework Algorithm Complexity

The AQOSA framework algorithm consists of two major parts: evaluation algorithm and optimization selection algorithm. The complexity of these parts cannot be directly compared as it is governed by different parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>the number of objectives (dimensions)</td>
</tr>
<tr>
<td>μ</td>
<td>the number of parents</td>
</tr>
<tr>
<td>λ</td>
<td>the number of offsprings</td>
</tr>
<tr>
<td>T</td>
<td>the length of simulation</td>
</tr>
<tr>
<td>n</td>
<td>the number of queues (number of nodes + number of buses) in the queuing network</td>
</tr>
<tr>
<td>p</td>
<td>the average number of events in the queue at times before the computed event</td>
</tr>
</tbody>
</table>

Table 4.2: Algorithm complexity parameters

In the evaluation part, the time complexity of simulating QN is in $O(m \cdot p)$, where $p$ is the average number of events in the queue at times before the computed event and $m$ is the number of events. If $T$ be the length of simulation and $n$ be the number of queues (number of nodes + number of buses) in the network, then we know that $m < T \cdot n$. Therefore, the time complexity of simulating QN is in $O(T \cdot n \cdot p)$.

In the optimization selection part, the complexity depends on the chosen algorithm. The time complexity of NSGA-II is in $O((\mu + \lambda) \cdot \log^{(d−1)}(\mu + \lambda))$ per iteration [BS14], where $d$ is the number of dimensions (objectives), $\mu$ is the number of parents, $\lambda$ is the number of offsprings. On the other hand, the complexity of SPEA2 is in $O((\mu + \lambda + A)^2 \cdot \log(\mu + \lambda + A))$ per evaluation, where $A$ is archive size. And finally, the time complexity of the selection step in SMS-EMOA in 2D and 3D is equal to $\Theta(\mu \cdot \log(\mu))$ [EF11] (although, incremental updates can be achieved faster if non-dominated sorting is replaced by a queuing method [HE13]). For higher dimensions, AQOSA does not support this algorithm due to efficiency problems. Note that SMS-EMOA performs a selection step for each new individual, while for NSGA-II and SPEA2 selection step is only done for any batch of $\lambda$ individuals.

Thereby as a result, if the system designer chooses the NSGA-II algorithm, the amortized complexity of running AQOSA per processed individual is in:

$$O\left(\frac{\mu}{\lambda} \cdot \log^{(d−1)}(\mu + \lambda) + T \cdot n \cdot p\right)$$

(4.1)
4.9 Summary

This chapter described the meta-heuristic optimization approach for automated software architecture design and its tooling which is developed to enable us for answering research questions in this dissertation. This approach offers a new tool for architects to aid in finding good designs in complex design situations with potentially conflicting multiple quality requirements. Furthermore, the tool reduces the development time and improves the quality of the architecture design. AQOSA framework supports multiple quality attributes for the optimization including response time, processor utilization, bus utilization, safety, and cost.

Inspired by the model-driven approach, the framework uses an integrated model (AQOSA IR) which helps performing multiple quality analysis based on a single core architecture representation. Moreover, this framework can be extended with additional quality attributes.

The approach has been applied on case studies which are described in next chapter. AQOSA framework improves over the state of the art because:

- It is modelling language independent. It can interoperate with various architectural modelling languages.

- It supports multiple degrees of freedom for automatically generating alternative architectures.

- It optimizes multiple quality attributes at once. To the best of our knowledge it is the first approach which supports evaluation and optimization of five quality attributes at the same time.