UNIVERSITY OF GOTHENBURG

Diagnosing and Healing Bottlenecks in Architecture Designs Automatically

Master of Science Thesis in Software Engineering and Management

NOUSHIN KHAKI

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, April 2013

Diagnosing and Healing Bottlenecks in Architecture Designs Automatically

NOUSHIN KHAKI

Examiner: MATTHIAS TICHY

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden April 2013

Abstract

Software architecting is one of the major phases in software development. A software architecture design should meet the desired functional and quality requirements of a system. Sometimes, the quality properties are in contrast with each other such as cost and CPU utilization which is achieved by a more expensive CPU. It makes hard for an architect to design architecture accompanied by optimizing all the desired quality properties. On the other hand, these days systems are getting more complicated by applying high technologies. Also, time to market is an important matter in developing a system. All these matters make architecting a difficult job. Hence, it is the time to make this job automated.

One of the main concerns in designing software systems is preventing problems by bottlenecks. It should be considered at the time of architecting. If it is ignored, it might be led to fundamental problems in the product. This research focuses on this problem and tries to find some approaches to solve it automatically in architecture designs.

Acknowledgments

I would like to thank my supervisor Michel Chaudron who directed me in a best way throughout my work. I had a great chance to complete my studies in the area of my interest, Software Architecting, which was his suggestion for my Master thesis. He guided me in every step perfectly, and let me have a feedback in a very short time. His enthusiasm to see the result made me so motivated for going ahead and making progress. Besides, he was quite understanding in the difficulties sometimes I had during the whole work. So, here I would like to be grateful of the head of IT University of Gothenburg and Chalmers, Agneta Nilsson, who introduced him to me as my supervisor.

Many thanks to Ramin Etemadi who is one of the excellent researchers in making automated software architecture designs. He helped me tightly to understand the framework he has made for this purpose, and directed me in extending and continuing his work. He was quite responsive to my several questions patiently.

It is my pleasure to work with them for a period of my life, and I really enjoyed doing this project as my Master thesis.

Table of Contents

1. Introduction

Architecture in software development context means the fundamental parts of a system including software and hardware components, the relationships among them, and the principles leading to design and development [17]. Bass et al. state software architecture is the structure of a computing system that includes software elements, their obvious properties, and the relationships between them [4].

While an architecture design prescribes the structure of a system involving the functional requirements, it represents the desired quality properties of a system as well. For instance, if a system needs high security, accessing to information should be managed in the time of architecting. Also, it is a common way for stakeholders to understand the system and negotiate about it [4]. Thus, it is an important matter in developing a system.

Nowadays, computer systems are being more complicated due to day by day innovation and technology advancement. The desired quality requirements of a system may be in contradiction with each other. For instance, considering high CPU utilization by a powerful CPU, which is expensive, is in contradiction with cost. Moreover, time-to-market is a significant feature to produce a system. Hence, it is a difficult job to design an architecture which meets all desired customer needs. Besides, the design needs to be optimized based on the available resources for a project. The resources dedicated to a project are usually restricted by time and budget. Although a system architect might be highly experienced, designing a high-tech system in a short time which is optimized would be hard. Thus, there is a need to make this job automated and save a lot of time and energy.

1.1. Problem Statement

One of the concerns in designing architectures is the existence of bottlenecks which are led to some problems in developed systems. Bottlenecks are created by the system resources such as hardware, software or bandwidth which restrict data flow or processing speed [19]. Nielson et al. relate them to performance constraints made by slow execution of a process [18]. They could impede response time and cause a performance problem. Woodside M. states bottleneck is a task that is totally utilized while the resources are not fully consumed [25]. Such cases increase CPU utilization or bus utilization which result low performance in systems. However, it is not limited only to performance quality attribute. They might appear against different quality attributes. Some bottlenecks affect the reliability of systems. If they make crucial situations, they may result a failure in the system. Therefore, it is important to realize them before happening, and fix them when they are appeared. The best way to prevent them is considering the predictable cases in the time of architecting.

One of the endeavors toward automated design of architectures is detecting and fixing bottlenecks automatically. The questions discussed in the research are as the following:

- How would bottlenecks be diagnosed?
- How would bottlenecks be healed?

This research focuses on CPU utilization, cost and reliability as the quality properties of a system. First, it addresses some alternatives to diagnose and heal bottlenecks relevant to the properties. Then, it makes the suggested solutions automated. The suggested solutions are implemented within AQOSA framework which is a tool for generating optimized architecture designs and will be described in section 3.1.

1.2. Related Work

Search Based Software Engineering (SBSE) is a research which contains efficient approaches to find optimal solutions for the problems in different phases of software engineering such as requirement engineering, project management, design, etc. [13]. Architecture optimization is a subset of SBSE. However, SBSE does not study the quality properties in detail. There are some other papers that analyze a certain area in designing an architecture concerned with a specific quality attribute or a domain system. For example, Grunske et al. [12] study on different approaches for optimization of safety in embedded systems. Kuo et al. [16] have published a survey for the systems that require high reliability. Villegas et al. [23] analyze self-adaptive systems with the aim of runtime architecture optimization. As it is stated, although they all offer alternatives for optimization problems, none of them present automated approaches.

Trubiani C. et al. have done some efforts to detect and solve performance problems in architectural models automatically [21]. They have studied the predictable performance problems in architecture models, and offered some alternatives to remove them from the models. Also, they have specified the alternatives that can be done automatically. However, it is just restricted to performance problems and not any other quality properties.

There are a lot of optimization researches based on Genetic Algorithms like [1] and [14]. Genetic Algorithms are used for optimization problems and will be described in section 2. One of the surveys creates a framework called AQOSA (Automated Quality Driven Optimization of Software Architecture) which optimizes architecture designs automatically [10]. It is built on the previous work done by Chaudron M. et al. [8] [9]. It applies Metaheuristic approaches, which is described in section 1.3, to find optimal solutions. The framework supports five quality attributes containing CPU utilization, bus utilization, response time, reliability and cost. The framework does not detect the predictable performance problems in architecture designs as Trubiani C. et al. have offered. It does not detect any other quality attribute problems, such as predictable bottlenecks related to cost or reliability, either.

This research tries to add some automated alternatives for other quality attributes (cost and reliability) to the research done by Trubiani C. et al. while it implements some of the offered solutions for performance problems. Also it tries to complement the automated optimization process of AQOSA by detecting and fixing bottlenecks related to performance, cost and reliability.

Aleti et al. categorize such researches based on their focus on problems, solutions or validations [2]. Since the context is extensive, every survey can be categorized under the defined taxonomies. The state of the art in this research is focusing on the problem of bottlenecks in architectures particularly concerned with these quality attributes: CPU utilization, cost and reliability. Moreover, it makes the suggested solutions automated within AQOSA.

1.3. Methodology

Generally, the method to carry out the whole work is categorized as the following:

- Collecting information to find some alternatives for diagnosing bottlenecks
- Analyzing the information to study which alternatives are possible to be automated
- Creating solutions to heal the bottlenecks automatically

Antipattern-based approach is used for diagnosing bottlenecks. Antipatterns are in the opposite of patterns that describe positive and constructive solutions. They focus on negative and destructive aspects of a system which result common problems such as bottlenecks, and describe common solutions to prevent them [6].

Then, the antipatterns should be analyzed to see which of them can be implemented to work automatically. Next, the ones that are possible to implement within AQOSA are selected.

Afterward, some solutions should be designed for fixing them. Antipatterns suggest some solutions, however; other solutions are created by coming up with new ideas. They are mixed with Metaheuristic-based approaches. Metaheuristic-based approaches are applied in solving

optimization problems. Metaheuristic is an iterative process which searches in a large space and designate some solutions in each iteration to optimize an object [7].
By applying the stated methods, some operators are designed for diagnosing and healing bottlenecks. The outcome will be a set of optimized architecture designs that lack the discussed bottlenecks.

1.4. Validation

In order to validate the designed operators which cure the bottlenecks in architecture designs, some experiments were carried out to demonstrate their functionality and efficiency to heal the bottlenecks. They will be described in "Experiments" section completely.

1.5. Outline

In the following sections, all the work done for this research is stated. Since Genetic Algorithms is the basis of optimization process in the research, it will be explained briefly in section 2. The whole work for diagnosing bottlenecks and healing them will be explained in section 3. First, there is an explanation about AQOSA. Then, the operators made for this purpose, which are embedded in AQOSA, will be described in detail. In section 4, all the experiments done for validating the operators are stated in addition to the achieved result. At the end, you will find a discussion around the work, and a conclusion containing future work.

2. Genetic Algorithms

Evolutionary Algorithms is a field of study associated with biology, Artificial Intelligence and optimization. The idea of it is inspired by the natural process of evolution. It was created with the aim of finding solutions for optimization problems. There are various forms of it such as Evolution Strategies (ES), Evolutionary Programming (EP) and Genetic Algorithms (GA) [3]. The one that is applied in this research is GA which is described in the following.
In biology, genes are single units which are strung together to form chromosomes within cells. In the process of evolving, the chromosomes are copied, go through crossover and mutated [11]. As GA suggests, a random population of solutions is generated. Each solution is a binary string called genotype which is a single unit in one generation. The next generation is evolved by selecting a number of genotypes, and performing some operators such as Crossover, Copy and Mutate on them. This process is repeated and continues up to a stop point which could be a certain number of generations or when the optimum solution is found [20].
The generated genotype is called offspring which is created by two parents. For example, two parents, which are shown by binary strings, are as the following [5]:

- Parent 1: 1101100100110110
- Parent 2: 1101111000011110

If "Crossover" is operated on them, they will generate new offsprings [5]:

Parent 1	11011 | 00100110110
Parent 2	11011 | 11000011110
Offspring 1	11011 | 11000011110
Offspring 2	11011 | 00100110110

"Mutate" is done by changing the binary codes randomly [5]:

Original offspring 1	1101111000011110
Original offspring 2	1101100100110110
Mutated offspring 1	1100111000011110
Mutated offspring 2	1101101100110110

Some of the most important properties of GA that make them quite suitable for optimization are as the following [5]:

- They work in parallel mode and search for good solutions in multiple directions simultaneously while most of the algorithms can search for a good solution in one direction at a time.
- They are good for the problems that their solution is in a huge space, and also complex. They could search in a vast space and successfully find good solutions.
- They could manipulate different parameters and work on them at the same time.

Since automated architecture design optimization needs to manipulate different parameters and optimizes different objectives simultaneously while the domain of solution is complex and extensive, it is concluded that GA can be an appropriate option for this problem.

3. Diagnosing and Healing Bottlenecks

This section describes how bottlenecks are diagnosed and how they are healed. As it was stated in "Methodology" section, some operators are designed to serve this purpose. In order to make the abstract models of the operators pragmatic, they are implemented. Furthermore, they are validated to demonstrate their functionality and efficiency. The implementation is done within AQOSA which is a framework that generates optimized architecture designs automatically. Hence, an explanation of AQOSA will come first in the following. Then, there will be a description about how it is extended by the operators. Afterward, the operators will be explained in detail. At last, there is a brief description about the designed classes and implementation detail.

3.1. AQOSA

AQOSA is a framework to generate optimized architecture designs automatically. It supports five quality attributes including CPU utilization, bus utilization, response time, reliability and cost. It has been designed based on Evolutionary Algorithms, specifically GA by using genetic operators such as Crossover and Mutate. According to the definition of the algorithm, there is a concept called genotype. In this context, it represents an individual architecture design. It is a string containing the related information. This is an example of a generated genotype by AQOSA:

[1, 2, 3, 3, 2, 3] [3|(43300.0, 525.0)(43300.0, 525.0)(40000.0, 350.0)] [3|(160.0, 3.0, 66.0)(128.0, 1.0, 60.0)(160.0, 5.0, 51.0)] [1, 1, 0; 1, 1, 1; 0, 1, 1]: [0.34774487067617854, 0.1521855602404377, 0.056287945557590466, 0.015759523339550843, 0.1577]

This sample is composed of 6 components and 3 nodes as it is interpreted by the following parts:

- [1, 2, 3, 3, 2, 3]: It states which components are deployed on which nodes, the array index represents the number of component and the array value represents the number of node: component 1 on node 1, component 2 and 5 on node 2, component 3, 4 and 6 node 3.

- [3|(43300.0, 525.0)(43300.0, 525.0)(40000.0, 350.0)]: It states there are 3 nodes, and each parenthesis represents the CPU properties of the nodes respectively. For instance, the CPU clock of node 1 is 43300.0 MHz, and the cost is 525.0 EUR.
- [3|(160.0, 3.0, 66.0)(128.0, 1.0, 60.0)(160.0, 5.0, 51.0)]: It states there are 3 buses, and each parenthesis represents the bus properties. The numbers in each parenthesis are band width, latency and cost respectively.
- [1, 1, 0; 1, 1, 1; 0, 1, 1]: It is a matrix for representing how the nodes are connected by buses. It is interpreted as the following

	Node No.1	Node No.2	Node No.3
Bus No. 1	1	1	0
Bus No. 2	1	1	1
Bus No. 3	0	1	1

  For instance, node 1 is connected to bus 1 and 2, but not 3.
- [0.3477448706761785\4, 0.1521855602404377, 0.05628794557590466, 0.015759523339550843, 0.1577]: They are values of response time, CPU utilization, failure probability, cost and bus utilization respectively, which are optimized by AQOSA.

Regarding the behavior of the algorithm, AQOSA generates a number of genotypes, and then designate some of them for optimizing and creating next generation. It is repeated up to a stop point which is the maximum number of generations or a criterion on the objective function. It gets some inputs, processes them and suggests a set of optimal architecture designs as an output. The inputs are as the following [10]:

1. Software components, that meet the functional requirements of the system, and their communications
2. A set of scenarios which demonstrate the work flow of the system
3. Objectives that state which quality attribute should be optimized, such as reliability or cost.
4. A repository of hardware and software specifications

Figure 1 shows the architecture of the framework [10].

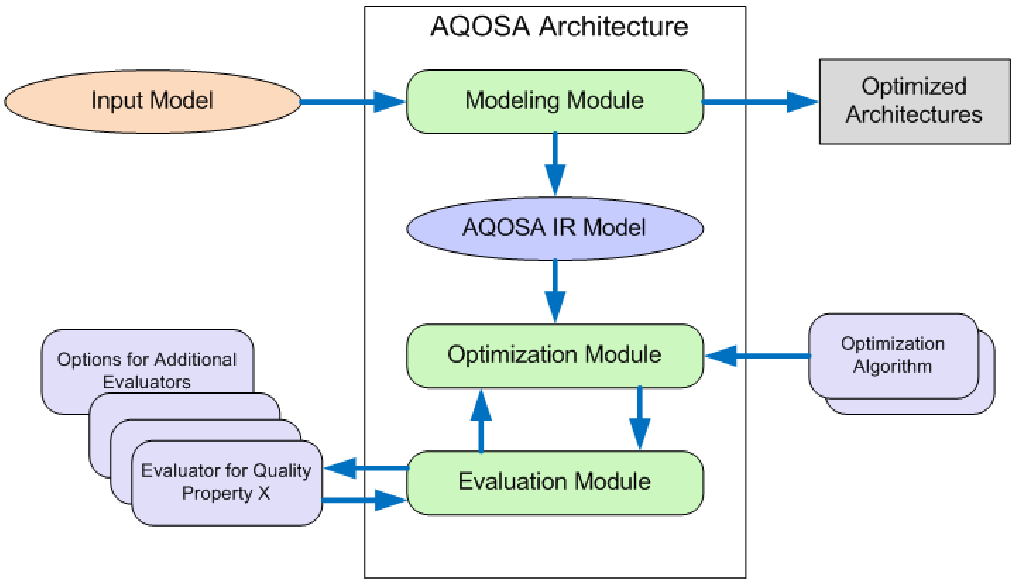


Figure 1: AQOSA overall architecture

As it is shown in the picture, there are three main modules that carry out the work:

1. Modeling module
2. Optimization module
3. Evaluation module

Modeling module gets input and converts it to AQOSA IR Model (AQOSA Intermediate Representation Model) to be understandable for Optimization and Evaluation modules. AQOSA IR Model is independent of any specific modeling language with the purpose of applying the framework in different domains [10].

Optimization module applies genetic operators such as Crossover and Mutate to optimize genotypes regarding the concerned quality attributes [10].

Evaluation module uses some evaluators, like a Fault Tree Analysis method which is applied for reliability, and evaluates the optimality status of quality attributes in genotypes.

As it is depicted in figure 1, Optimization module and Evaluation module work iteratively on the models to generate the optimal result. The output will be optimized architectures represented by genotypes [10].

3.2. The Extension work

As it was stated in the previous section, AQOSA generates a number of genotypes which represent architecture designs. The generated genotypes may contain some defects which probably appear as the bottlenecks in developed systems in the future. In order to remove the defects, it is required to check the health of each genotype, and heal the defects when they are appeared.

Optimization module, which works based on GA, involves genetic operators such as Copy, Crossover and Mutate [10]. First, it generates a number of genotypes randomly. Then, it selects some of them to be operated by the genetic operators. As it was described in “Genetic Algorithms” section, two parents are needed to be operated by the genetic operators and generate two offsprings. The offsprings are sent to the new operators for diagnosing any bottleneck. If there is any bottleneck in offsprings, they will be healed. As it is described, the new operators should work tightly with genetic operators. Hence, they are considered to be embedded in Optimization module.

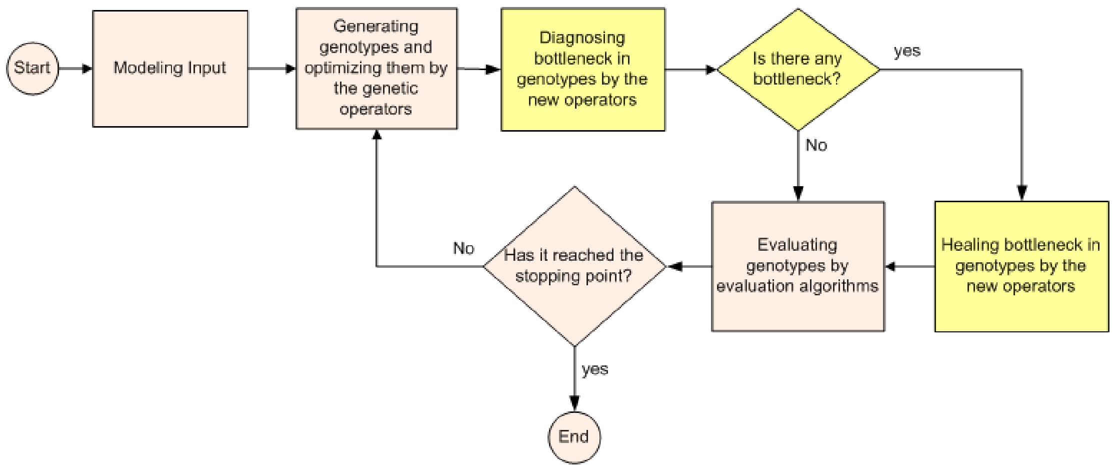
Subsequently, the offsprings are sent to the Evaluation module. Evaluation module evaluates them from the objective point of view by the evaluation algorithms. If the process should be ceased by the stopping point condition, it is ended. Otherwise, the offsprings will be sent to Optimization module and the process goes on.

Figure 2 is a flow chart that shows the flow of the work. The yellow boxes show the extended work. As it is shown, the new operators work right after optimizing genotypes by the genetic operators and before evaluating by Evaluation module.

3.3. Antipatterns

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 [22].

Since bottlenecks are the result of negative features in software systems, antipatterns are studied in order to diagnose them in a proper way. Further, some appropriate solutions for healing them are developed. The suggested solutions are based on the antipatterns solutions, also by coming up with new ideas. In result, some operators are designed and developed to serve this purpose. The operators will be described in section 3.4.



Start
Modeling Input
Generating genotypes and optimizing them by the genetic operators
Diagnosing bottleneck in genotypes by the new operators
Is there any bottleneck?
yes
No
Healing bottleneck in genotypes by the new operators
Evaluating genotypes by evaluation algorithms
Has it reached the stopping point?
No
yes
End

Figure 2: Flow of the operators between optimization and evaluation

There are antipatterns stated in [22]. However, it is not possible to consider all of them. Some of them can be converted to automated solutions, and among them, just some can work within AQOSA framework due to its restrictions. According to the current design of the framework, the operators that diagnose and heal bottlenecks are limited to the following changes:

- Software component replacement
- Hardware component replacement
- Communication lines replacement
- Software on hardware allocation
- Network topology

Therefore, if an antipattern describes changes in software components such as redesigning some classes or data structure, they could not be included in this research. There are two antipatterns studied in this research which will be described in the following.

3.3.1. Concurrent Processing Systems

As it is stated in [22], "it occurs when processing cannot make use of available processors". It means that 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 [22]. Figure 3 illustrates the problem by an example. "t" represents execution time of each component.

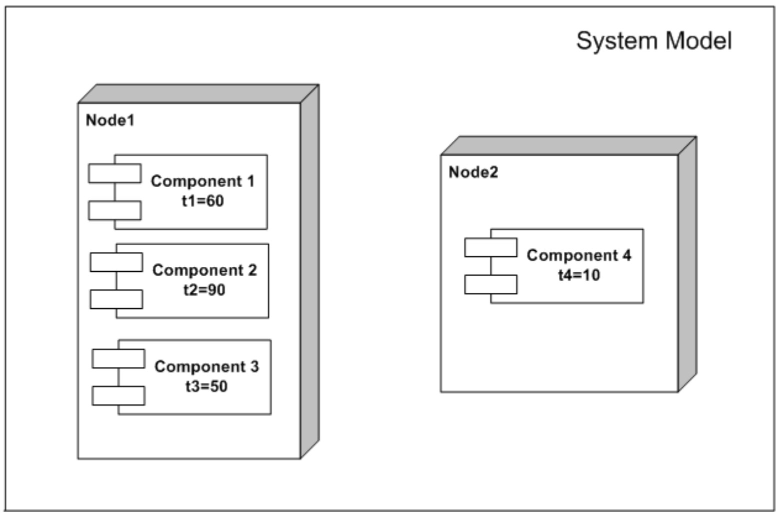


Figure 3: Concurrent Processing Systems Problem

The example shows that there are three components on node 1 whereas there is just one component on node 2. Also, execution time for component 4 on node 2 is 10s which is much less than the execution time of the components on node 1 which is 200s (t1+t2+t3= 60+90+50). It is clear that the software components are not assigned to the hardware in a balanced way. The CPU utilization in node 1 would be high while it would be low in node 2. Regarding the definition of bottleneck by Nielson in section 1.1, high CPU utilization of node 1, which makes the execution of the process slow, could result a bottleneck. In addition, according to the statement by Woodside in section 1.1, node 2 is not fully consumed by component 4, so it could be led to a bottleneck as well.
A suggested solution by [22] is "restructure software or change scheduling algorithms to enable concurrent execution". It recommends reorganizing deployment of the software components in a better way. So, regarding the available resources, the components should be redeployed in a balanced way [22]. Figure 4 shows a sample of balanced assignment of the processes to the resources.

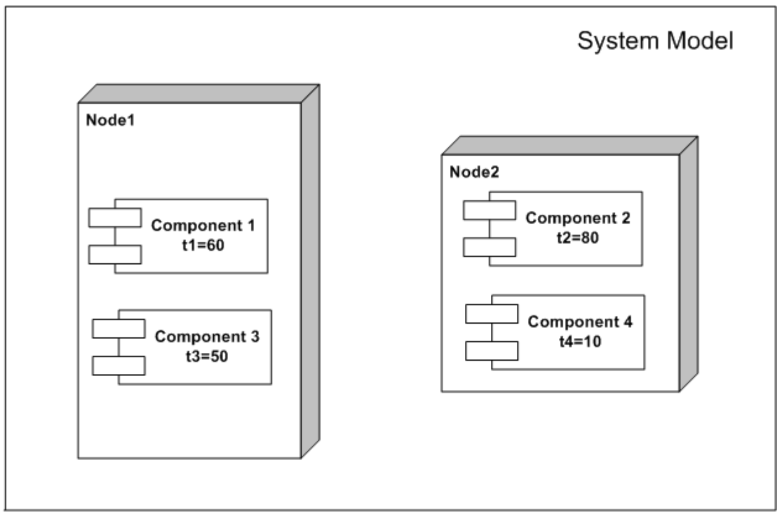


Figure 4: Concurrent Processing Systems Solution

As it is seen in the picture, component 2 has been moved to node 2. The execution time of the component in node 1 was 90s while it could change after moving to node 2 due to the power of CPU. In this example, we suppose that it has been less than before (80s). In result, there are two components on node 1 with the execution time of 110s (60+50), and two other components on node 2 with the execution time of 90s (80+10). As it is demonstrated, the execution time of node 1 is reduced to 110s. Therefore, when the two nodes work concurrently, the whole execution time of the system will be less than before which result a better performance.
The suggested solution can be applied automatically in AQOSA framework. There are more solutions which will be introduced in "Operators" section.

3.3.2. Pipe and Filter

As it is stated in [22], "it occurs when the slowest filter in a Pipe and Filter architecture causes the system to have unacceptable throughput". It means that there is a phase in a pipe and filter system which takes time to be completed, so it makes the whole system slower to generate the outcome [22]. Figure 5 shows a simple Pipe and Filter architecture.

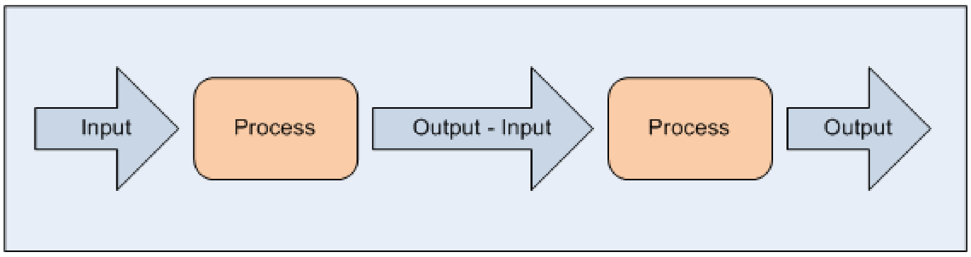


Figure 5: Pipe and Filter Architecture

The solution that [22] suggests is “Break large filters into more stages and combine very small ones to reduce overhead”. It means dividing big processes, that has long execution time, to multiple small processes, that has short execution time, and run them in parallel. The processes should be combined at last. The result will reduce the overhead of the system [22]. It is demonstrated in figure 6.

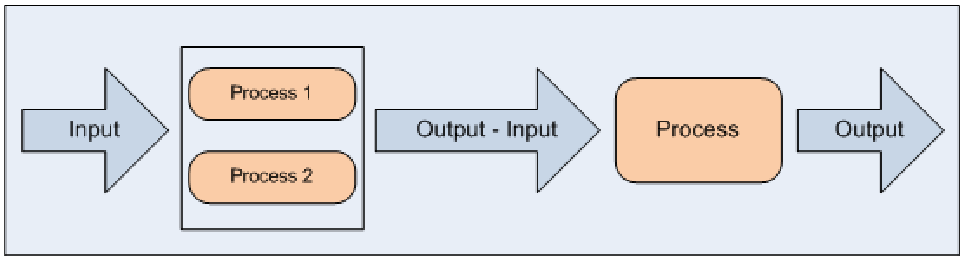


Figure 6: Pipe and Filter Solution

Since it is not possible to change the software components in AQOSA framework and break them into small pieces, the solution is changed in a way that it could be possible to implement within the framework. It will be described in section 3.4.6.

3.4. Operators

In order to diagnose and heal bottlenecks automatically, some operators are designed and developed. The design is based on the introduced antipatterns. They are as the following:

1. Component Movement: to remove probable bottlenecks for CPU utilization.
2. CPU Change for Performance: to remove probable bottlenecks for CPU utilization.
3. CPU Change for Cost: to remove probable bottlenecks for cost.
4. CPU Change for Reliability: to remove probable bottlenecks for reliability.
5. Load Balancer: to remove probable bottlenecks for CPU utilization.

The first one is implemented totally based on “Concurrent Processing Systems” antipattern and the solution it suggests. The idea of making second and third one is got from the same antipattern when there is a node with high or low CPU utilization in the system, however; the solutions are new ideas. The forth one is designed entirely by a new idea. Load Balancer is a simulation of “Pipe and Filter” antipattern. The following sections explain about the functionality of the operators. Beforehand, there is an explanation about “threshold” which is an essential criterion for every operator.

3.4.1. Threshold

As it was stated earlier, the quality properties, which are considered in this research, are CPU utilization, cost and reliability. Every operator needs to define a threshold in order to find high and low values of each quality property that cause a bottleneck. For instance, if an operator is designed to detect bottlenecks caused by high or low CPU utilization, there should be a threshold in that every CPU utilization above or below it represents high or low CPU utilization which cause the bottleneck. If a high limit for CPU utilization in a system is 80%, threshold will be 80%, and it means that every utilization higher than 80% is led to a bottleneck. Also, if a low limit for CPU utilization is 20%, threshold will be 20%, and it means that every utilization lower than 20% causes a kind of bottleneck (not using the whole resources of a system). It is similar for failure probability of a CPU which is related to reliability attribute. For example, if a high limit for failure probability is 20%, threshold will be 20%, and every failure probability higher than it is led to a risk of failure for the system, and is a crisis for reliability.

Since different systems have different resources and properties, threshold is differed from one model to another one. Moreover, there is not just one threshold for all the operators in a system. Due to the goal of each operator and the method it uses, threshold is defined for every one individually.

AQOSA offers a set of optimized architectures with optimal values for the quality properties it involves. When operators are added to it, they try to reduce these optimal values. For instance, Component Movement tries to reduce CPU utilization. A right threshold for this operator is the one that helps to decrease utilization value. Similarly, the right threshold for cost is the one that helps to decrease cost, and the right threshold for reliability is the one that assists to reduce failure probability.

It is achieved by try and error. First, every single operator is set to AQOSA. A number is chosen as a threshold for it. The result will be studied to see if they are decreased. Then, it is changed dependant to its efficiency. After trying some numbers, the one that makes a significant drop for the value of a quality property is selected. However, when the operators are combined to work together, the threshold may not be as good as they work individually. Hence, two or three good threshold is chosen for every operator when they are examined individually. Then, they are examined again after combining the operators. At last, the ones that could make drop for quality attributes are chosen.

3.4.2. Component Movement

According to “Concurrent Processing Systems” antipattern, non-balanced assignment of processing to processors can make the system slow and cause a performance bottleneck. Figure 7 shows a sample system with four nodes associated with the anipattern. “t” represents execution time of each component.

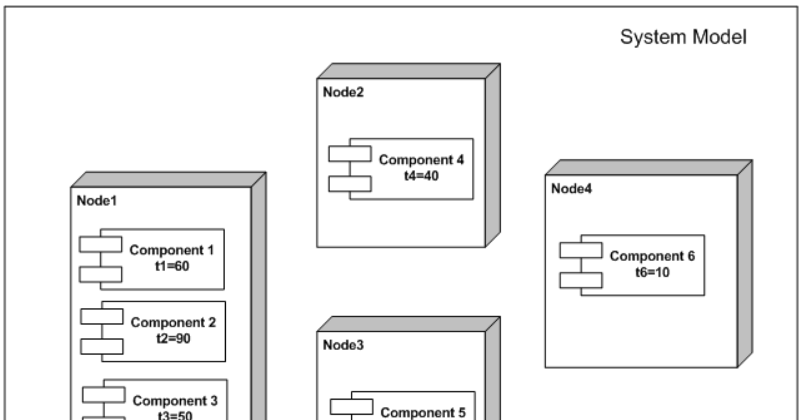


Figure 7: Non-balanced Assignment of Components to Nodes

As it is seen, there may be a node (node 1) in a software system containing some components which cause a high utilization. On the other hand, there may be a node (node 4) with just one component. So, the CPU utilization of the node would be low, and the whole resources are not fully used. As the antipattern suggests, a rearrangement of allocating processes to available resources is needed. The components on node with high CPU utilization should be moved to the node with low CPU utilization. To serve this purpose, two methods collaborate: diagnose and heal.
For diagnosing high and low utilization two thresholds should be defined for CPU utilization: high threshold and low threshold. They are defined as it was explained in section 3.4.1. The algorithm for diagnosis method is as the following:

1. Search for the maximum and minimum CPU utilizations relevant to an individual architecture design. (In this sample, node 1 has maximum utilization, and node 4 has minimum utilization.)
2. Compare the maximum utilization with high threshold, if it is greater than the threshold, then mark it as a node with high CPU utilization.
3. Compare the minimum utilization with low threshold, if it is less than the threshold, then mark it as a node with low CPU utilization.
4. If an architecture design contains a node with high CPU utilization and a node with low CPU utilization, then bottleneck is diagnosed.

CPU utilization is calculated in AQOSA. As it was stated in section 3.1, the framework gets a set of scenarios as input. It simulates a network of the nodes, and creates some events at a proper time by the predefined scenarios. Then, it calculates the utilization for each CPU based on this simulation, and sends them to the operator.
The heal method works as the following algorithm:

1. Calculate the execution time for each component: t1, t2, t3, t4, t5, t6. It is calculated by dividing the number of component cycles to CPU clock: N(component cycle)/CPU clock).
2. Calculate the average execution time for the nodes with maximum and minimum utilizations. In this sample, they are node 1 and 4 respectively. Thus, the average is: (t1+t2+t3+t6)/2 = (60+90+50+10)/2 = 105s.
3. Choose a component from the node with high CPU utilization and calculate the execution time of it on the node with low CPU utilization. In this sample, component 1 is chosen; and the execution time on node 4 is supposed to be 80s (see figure 8).
4. If the sum of the execution time on node with low CPU utilization is less than the average, then move the component, otherwise it is not right to move the component. In this sample, the sum of execution times of component 1 and 6 on node 4 is t1+t6 = 80s+10s = 90s, and it is less than 105s, so it is right to move (see figure 8).

The ideal case is when the execution time of both nodes (nodes with high and low utilizations) are approximately equal, and this equal value is defined by the average of them. It can be described as a balance instrument when the weights are divided in both scales equally. For instance in this sample, if node 1 takes 200s (t1+t2+t3=60s+90s+50s) while node 4 takes 10s to complete, by assuming that the nodes work in parallel, 200s is needed to complete the whole process. However, the ideal case is when the execution time for both nodes is near 105s which is the average ([200s+10s]/2). It is done by moving the components from node 1 to 4 which makes a drop for execution time of node 1. As it is seen by this sample in the balanced assignment, the sum of execution time on node 1 would be 140 (t2+t3

= 90+50), and the sum of execution time on node 4 would be 90s (t1+t6 = 80+10). By assuming that they work in parallel, 140s is needed to complete the whole process. The balanced allocation of the components to the nodes is illustrated in figure 8.
The implemented operator named "ComponentMoveCPSImpl" is shown in class diagram, figure 12. It is set in Optimization module of AQOSA which will be described more in section 3.5.

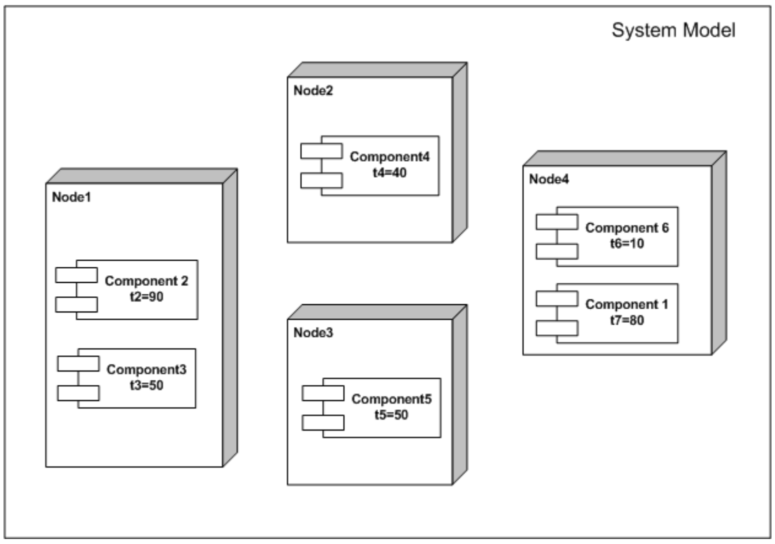


Figure 8: Balanced assignment of components to the nodes

3.4.3. CPU Change for Performance

When there is a node with high CPU utilization in the system, an alternative to reduce utilization could be replacing a better CPU instead of the current one. In AQOSA, there is a repository of available hardware resources. So, there are some options to choose. A CPU with greater clock would be more powerful and can reduce utilization, so it can be selected for replacement.
The operator involves two methods same as Component Movement: diagnose and heal.
For diagnosing high utilization, a threshold should be defined for CPU utilization. It is defined as it was described in section 3.4.1. Diagnose method searches for the maximum CPU utilization of the nodes in an individual architecture design. If it is greater than the threshold, then bottleneck is diagnosed.
Heal method searches for a CPU with greater clock in the repository, and replaces the CPU of node with high utilization by it. Figure 9 illustrates an example of changing CPU for the aim of decreasing utilization. As it is seen, when the CPU clock is 233GHz the utilization is 70%. After changing CPU by one with the clock of 333GHz, utilization is reduced to 50%.
The implemented operator named "CPUChangePerformanceCPSImpl" is shown in class diagram, figure 12. It is embedded in Optimization module of AQOSA which will be explained more in section 3.5.

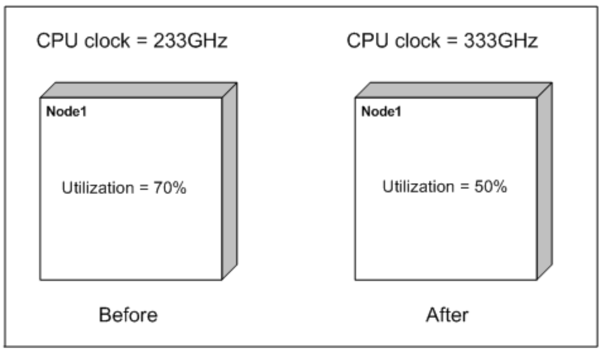


Figure 9: Changing CPU to decrease utilization

3.4.4. CPU Change for Cost

The former operators consider CPU utilization, and try to reduce it. However, cost is one of the quality requirements that is important at the time of designing the architecture of a system. Replacing better CPU with greater clock, which is more expensive, to decrease utilization makes an increase for cost. So, it should be decreased by a right approach.

If there is a node in a system which has low utilization, it will not need to have a powerful CPU with a great clock which is expensive. Hence, by identifying it and replacing a cheaper CPU, cost will be reduced efficiently.

Similar to the former operators, the operator for decreasing cost contains two methods: diagnose and heal.

For diagnosing low utilization, a threshold should be defined for CPU utilization. It was stated in section 3.4.1. Diagnose method searches for the minimum CPU utilization of the nodes in an individual architecture design. If it is less than the threshold, then bottleneck for cost is diagnosed.

Heal method looks for a cheaper CPU in the repository, and replaces the CPU of node with low utilization by it. Figure 10 shows an example that a cheaper CPU, which is less powerful, is replaced.

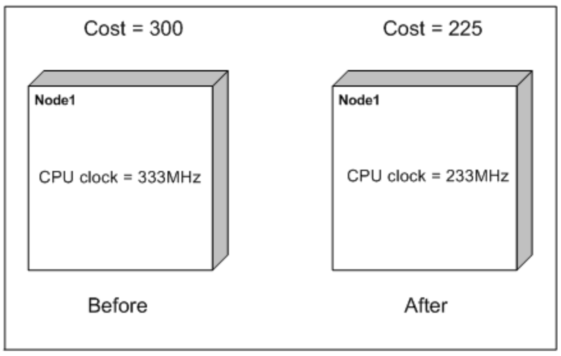


Figure 10: Changing CPU to decrease cost

If the cheaper CPU has the lower clock, it will not be an issue for CPU utilization overall. When the operators are set in the optimization process of AQOSA, each one tries to optimize the quality attribute that is responsible for. Since the optimization process is based on Genetic Algorithms which searches for good solutions in multiple directions at the same time, the result will be the optimum for all quality attributes.

The implemented operator named “CPUChangeCostCPSImpl” is shown in class diagram, figure 12. It is set in the optimization process of AQOSA which will be explained more in section 3.5.

3.4.5. CPU Change for Reliability

This operator is designed to decrease failure probability, and consequently increase reliability. There may be some nodes in an individual architecture design which has the CPU with high failure probability. They should be identified and replaced by the CPUs with lower failure probability. To serve this purpose, two methods cooperate: diagnose and heal.

To diagnose high failure probability, a threshold should be defined for failure probability. It is defined as it was explained in section 3.4.1. Diagnose method tries to find the nodes that are not quite reliable. To assess this measure, it gets the failure probability of each CPU, and compares it with the threshold. If it is greater than the threshold, then bottleneck is diagnosed.

Heal method changes the CPU of the node which is likely to fail. A good choice for reliability is defined based on the available resources in the repository. Absolutely, it would be a CPU

with lower probability to fail. CPUs with high failure probability are replaced by it. In result, reliability is increased.
The implemented operator named "CPUChangeReliabilityCPSImpl" is shown in class diagram, figure 12. It is embedded in the optimization process of AQOSA which will be explained more in section 3.5.

3.4.6. Load Balancer

The idea of making this operator is derived from "Pipe and Filter" antipatten. The approach needs to be changed due to the restrictions of AQOSA framework. If there is a component with high execution time, it is not possible to break it into small pieces in the framework. Instead, it is copied to another node which receives the requests same as the main one. Then, requests can be divided into two parts and sent out to two same components which process them at the same time. In result, there should be a decrease in response time. Figure 11 shows the suggested solution.

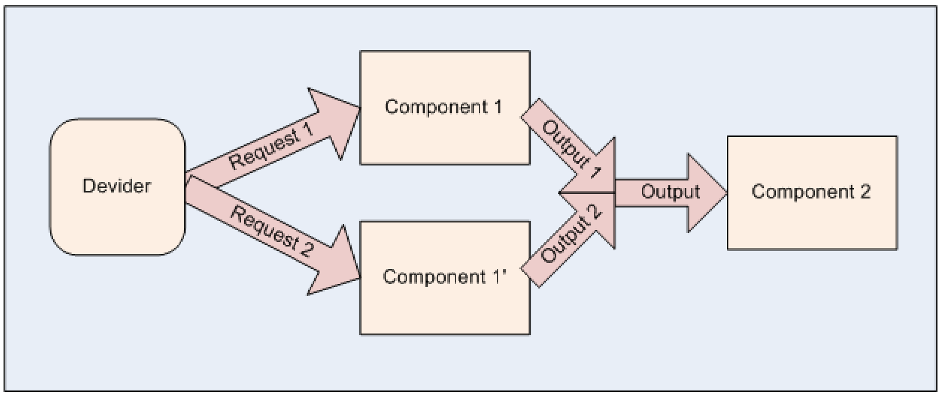


Figure 11: Suggested Solution for Pipe and Filter Problem

The operator has two methods like the other operators: diagnose and heal.
To diagnose high execution time, a threshold should be defined for each component individually due to its cycles. The execution time is calculated by dividing number of cycles of a component to CPU clock of the node that component is deployed on it. Diagnose method searches for the components in an architecture design that have high execution time. If there is any component with an execution time greater than its threshold, a bottleneck is detected.
Heal method copies the component to the node with minimum utilization in the architecture. It is expected that the response time is decreased in the optimization process of the framework. Unfortunately, it makes response time worse in the process of optimization! Since it is not possible for other changes within AQOSA such as decreasing requests, the simulation is ended at this point. So, this operator is not added to the framework as a useful one. The result of it is stated in the appendix, section 8.5.

3.5. Development Detail

AQOSA framework has been implemented based on Opt4J which is an optimization framework for applying Metaheuristic algorithms written in Java. So, the extended work is developed based on it by Java programming language.
Figure 12 is a class diagram that shows the extended work inside Optimization module. Yellow color represents new classes added to AQOSA.
"ConcurrentProcessingSystems" is an interface designed based on the antipattern with the same name. It contains the signature of diagnose and heal methods. It is connected to Optimization module by inheritance relationship to "ArchOperator" of Opt4J. It is

implemented first by “ConcurrentProcessingSystemsImpl”, which contains some common useful methods such as finding maximum or minimum CPU utilization, and then by the four operators: “ComponentMoveCPSImpl”, “CPUChangePerformanceCPSImpl”, “CPUChangeCostCPSImpl” and “CPUChangeReliabilityCPSImpl”, which involve different methods to diagnose and heal bottlenecks.
“ArchMatingModule” is a class that inherits from “Opt4JModule” class. It binds AQOSA framework to use “ArchMating” class instead of “MatingCrossoverMutate” class of Opt4J. “ArchMating” inherits from “MatingCrossoverMutate”, and uses the new operators to change the genotypes. The realization relation to the new operators demonstrates it.

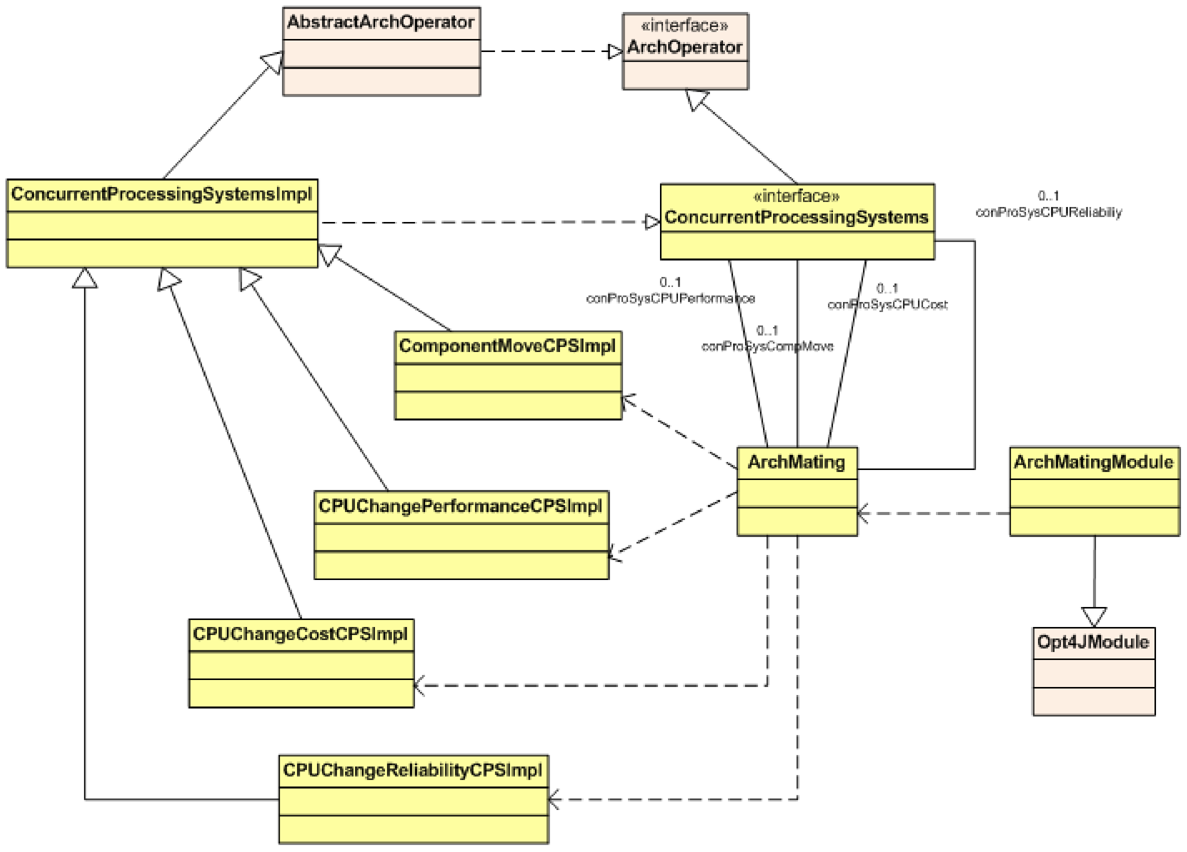


Figure 12: Class diagram

4. Experiments

Due to validating the operators, they are required to be examined. Also, in order to demonstrate the efficiency of them on the framework, different experiments need to be done. First, each operator is added individually to the framework, and consequently examined to expose its usefulness. Then, they are combined and work together, and accordingly examined to see if they are efficient as much as they work individually.
The input model for AQOSA contains 18 components. They represent system functionalities. Figure 13 shows how they are connected to each other.

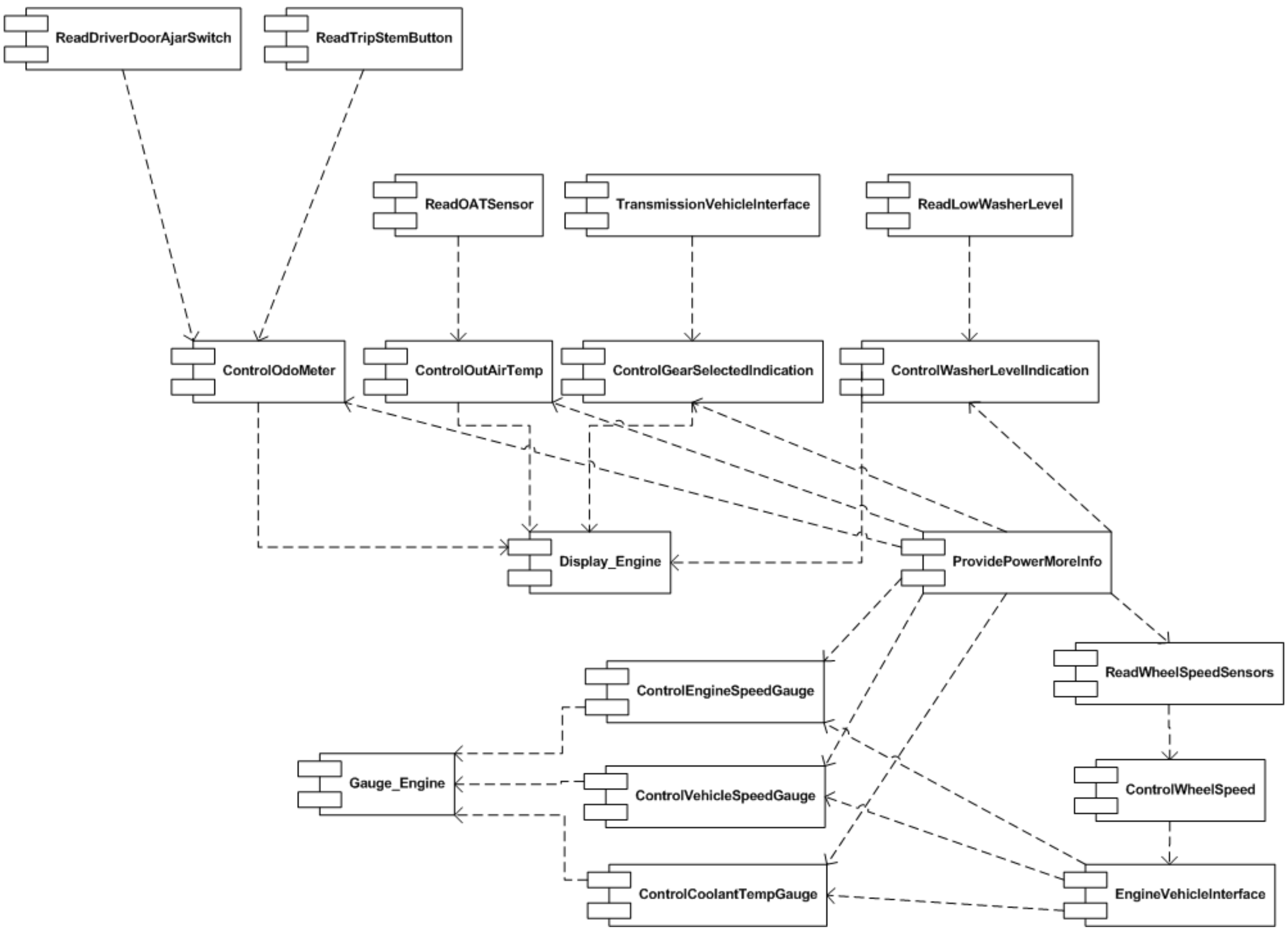


Figure 13: Input Model Component Diagram

The input model includes of 6 nodes which are connected to each other through communication buses. Figure 14 shows the nodes and the communication buses between them.

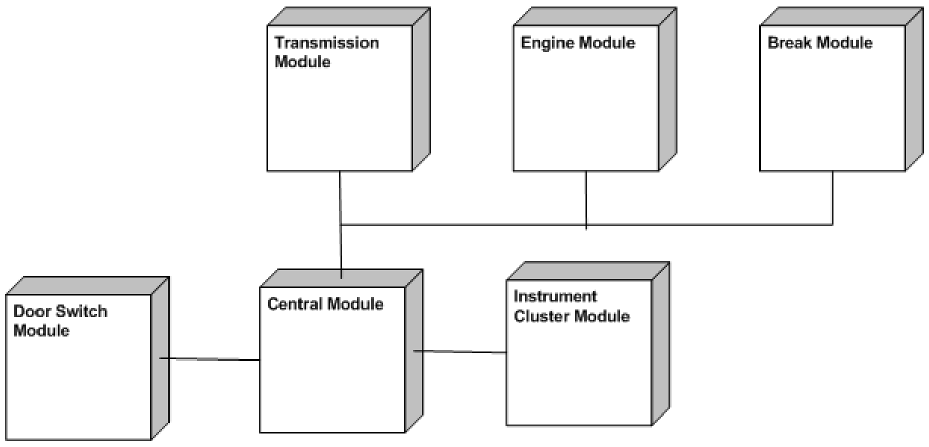


Figure 14: Input Model Nodes and Communication Buses

The properties of the nodes and buses are described in the following tables:

Node	Cost (USD)	Processor Speed (MIPS)	Lower Failure Rate	Upper Failure Rate
Break Module	100	80	0.01	0.025
Central Module	50	60	0.01	0.025
Door Switch Module	15	10	0.02	0.03
Engine Module	120	100	0.01	0.025
Instrument Cluster Module	50	60	0.01	0.025
Transmission Module	50	40	0.015	0.03

Table 1: Node Properties

Bus	Cost (USD)	Bandwidth (kbps)
HS CAN (cost/module)	1	500
LS CAN (cost/module)	0.25	33
LIN (cost/module)	0.1	10

Table 2: Bus Properties

There is a repository of hardware and software components. The software components are different implementation of the main components. The hardware components are:

- 28 processors: the processing speed varies from 10 MIPS to 100 MIPS. Each one has two levels of failure rate. A processor is more expensive if it has more processing speed or lower failure rate.
- 4 buses: the bandwidths are 10, 33, 125 and 500 kbps, and the latencies are 50, 16, 8, and 2 ms. A bus is more expensive if it supports higher bandwidth.

AQOSA gets the input model as an XML file. It is presented in appendix, section 8.7
The quality properties that should be optimized are given to the system as the objectives. As it was stated earlier, AQOSA involves five quality attributes for the optimization purpose: CPU utilization, bus utilization, reliability, response time and cost. Since the new operators are designed to remove bottlenecks related to CPU utilization, cost and reliability, they are set in AQOSA, and the others are removed.
The input for the operators involves genotypes which are studied with the aim of diagnosing and healing bottlenecks, also CPU utilization of the nodes.
AQOSA generates a Pareto plot as the output based on every quality property and the iteration which is the number of generation in the process of optimization. A sample of this plot is illustrated in figure 15 for cost. The total number of generations in this example, and also in all examinations is 200.
As it is shown in the picture, the blue line is considered for minimum values. The optimum point is where it reaches the lowest amount and remains steadily. The value of quality property and the number of iteration in this point is recorded. Since it is not right to rely on eyes and read the numbers on the plot, they are recorded in a log file. The value of optimum quality property is the main value to consider. It should be recorded in two types of examination: AQOSA *without* the new operator(s) and AQOSA *with* the new operator(s). Since the operators are considered to reduce the relevant quality property value, the result of the optimum value with the new operator(s) should be less than the result without the operator(s). On the other hand, if the number of iteration in AQOSA with the operator(s) less than it in AQOSA without the operator(s), it will be a good point, because it demonstrates that AQOSA becomes faster to reach the optimum point. If it is increased, it will not be an issue, because the goal of experiments is reducing the value for the quality properties.

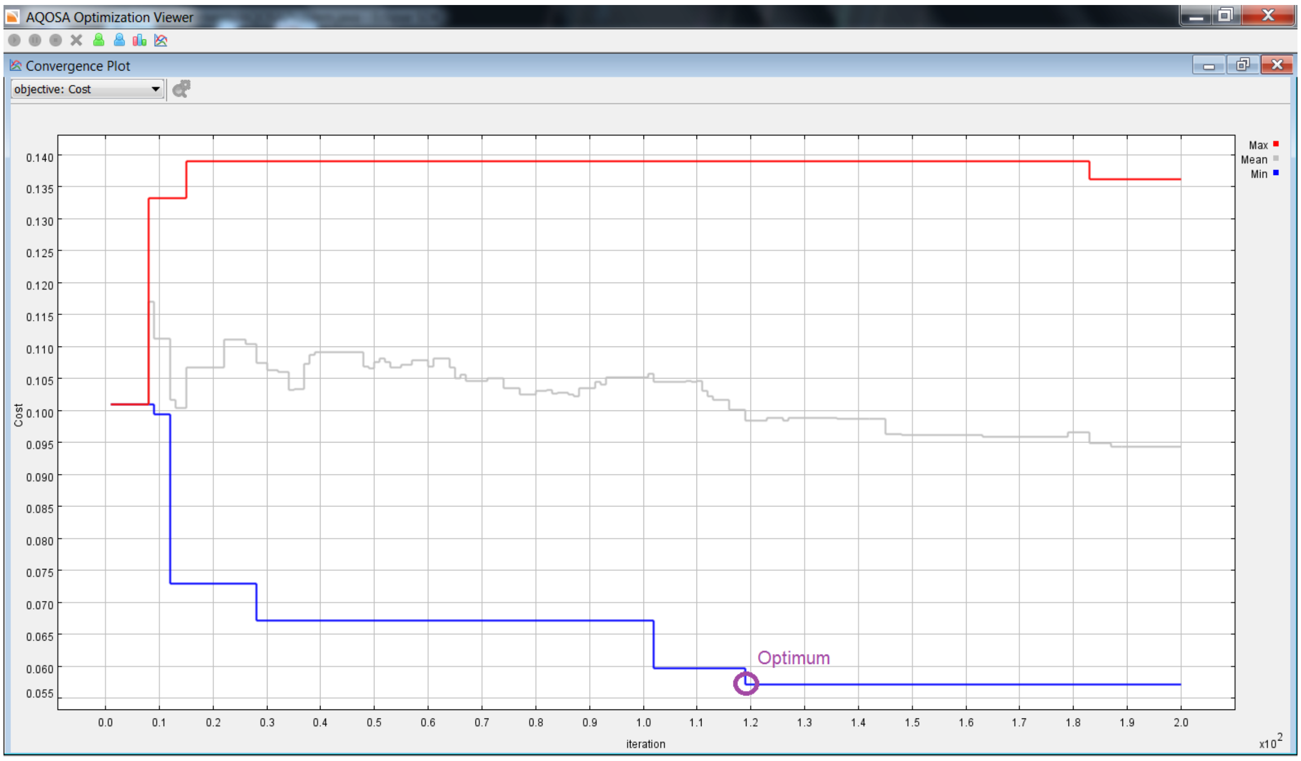


Figure 15: Pareto Plot

As the value of optimum point varies from one execution to another, the experiments need to be done several times in order to see whether they goes up or comes down. Thus, the experiments are repeated for 10 times in each case (with and without the operators). In result, there will be 10 numbers related to each quality property value. In order to compare these two sets of data, some statistical ways are required to be applied. Box plot [24] and t-test [15] are used for this purpose. Box plot is used to show if the data is increased or decreased by adding the operators. Then the significance level of the increase or decrease is demonstrated by t-test.
Box plot is a tool for presenting the range and distribution of a group of data. It uses 5 indexes: minimum, first quartile, median, third quartile and maximum to demonstrate the spread of data. These indexes are the basis for comparing two or more box plots which are representatives for sets of data [24]. Figure 16 shows a simple example of a box plot for the numbers of 1 to 5 (1, 2, 3, 4, 5).

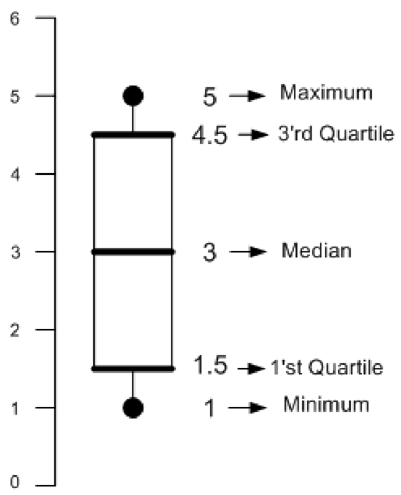


Figure 16: Box Plot

T-test is used to test the difference between two groups of data. There are different kinds of t-test related to comparing two different groups or the same groups at two different periods of time, etc. Since the groups in this study are different, independent samples t-test is used. First a "t" value should be calculated by the following formula [15]:

$$t = \frac{\bar{X}1 - \bar{X}2}{\sqrt{\left[\frac{SS1 + SS2}{\mathrm{n}^2 - \mathrm{n}}\right]}}$$

- $\bar{X}1$ and $\bar{X}2$ are the means of each group.
- n is the sample size of each group
- SS is sum of squares and is calculated by this formula: $SS = \sum x^2 - \frac{(\sum x)^2}{n}$

Then "t" value should be compared with the critical *t* value from a *t* table. To find the critical *t* value, the degree of freedom should be calculated as the following [15]:

- df = 2n-2

Then based on the value of *df* and the probability level presented in *t* table, the critical *t* value is found. If "t" value is greater than the critical *t* value, then we can conclude that there is a significant difference between two groups, otherwise there would not be a significant difference at the chosen probability level [15].

Therefore, two box plots are made for 10 optimal quality property values in the cases of AQOSA with and without the operators, and the indexes are compared to see if there is a decrease after adding the operators. Then, t-test is done to find whether the decrease is significant or not. If there is a significant reduction, then it will be concluded that the operators are useful. The same is done for iterations in order to see if the operators make the framework faster or not. The following sections describe the result by every operator individually and finally by the combined operators.

4.1. Component Movement

The objective of this operator is reducing CPU utilization. In order to find this matter, two sets of experiments are done for the cases of AQOSA with and without the operator. In each case, 10 values for the total CPU utilization of the system offered in the optimized architecture are recorded. They are presented in table 18 (appendix, section 8.1). Then, two box plots are made in order to compare the result as it is presented in figure 17. As it is seen in the picture and by comparing indexes presented in table 3, it is concluded that CPU utilization is decreased after adding the operator.

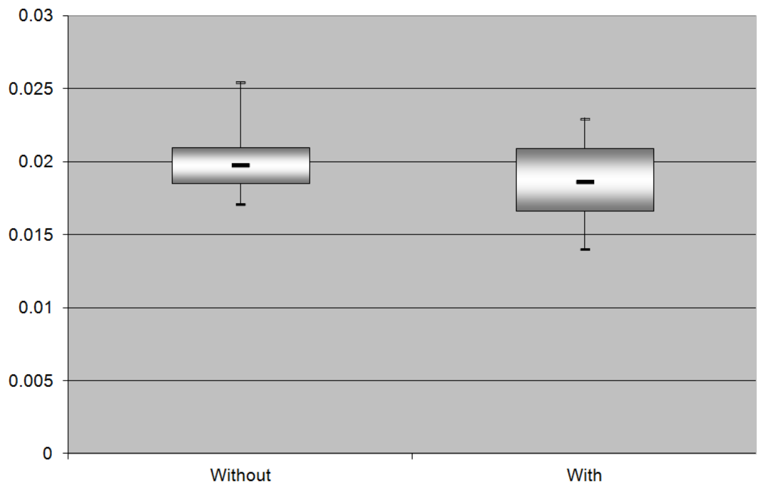


Figure 17: CPU Utilization Box Plot for Component Movement Operator

	Without	With
Minimum	0.017091396	0.014026414
First Quartile	0.018543671	0.016612936
Median	0.019827031	0.018661963
Third Quartile	0.020975818	0.020949402
Maximum	0.025484816	0.022981467

Table 3: CPU Utilization Indexes for Component Movement Operator

In order to see whether the decrease is significant or not, a t-test should be done. The value of "t" is calculated based on the data in table 18 (appendix, section 8.1) as the following:

$$\overline{X}1 = 0.020053508,\ \ \overline{X}2 = 0.018754217$$

$$SS1 = 0.004080503 - \frac{0.040214316}{10} = 5.90717\text{E-}05$$

$$SS2 = 0.003595529 - \frac{0.035172067}{10} = 7.83222\text{E-}05$$

$$t = \frac{0.020053508 - 0.018754217}{\sqrt{\left[\dfrac{5.90717\text{E-}05 + 7.83222\text{E-}05}{10^2 - 10}\right]}} = 1.051583082$$

$$Df = (2 \times 10) - 2 = 18$$

Looking at *t* score table for finding the critical *t* value with df=18, it is seen that the calculated "t" value is greater than the critical *t* value at 0.4 probability level:

$$1.051583082 > 0.862$$

Therefore, it is deduced that there is a significant decrease after adding the operator at 0.4 probability level.

The values for iteration are recorded for the two stated cases and presented in the box plots. The index values in figure 18 and table 4 show that the iteration is decreased by the operator.

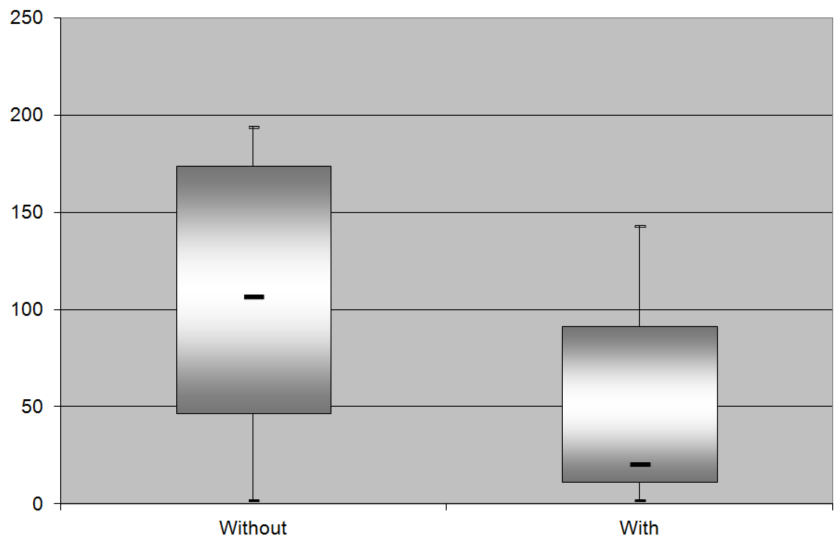


Figure 18: Iteration Box Plot for Component Movement Operator

	Without	With
Minimum	2	2
First Quartile	46.5	11.25
Median	107	21
Third Quartile	173.75	91.5
Maximum	194	143

Table 4: Iteration Indexes for Component Movement Operator

Same as what is done for the CPU utilization, the "t" value is calculated for iteration based on the data presented in table 19 (appendix, section 8.1) to see if the decrease is significant or not. The result is 1.959445216. Comparing it with the critical *t* value got from *t* score table, it is deduced that there is a significant decrease at 0.3 probability level:

$$1.959445216 > 1.067$$

Therefore, the operator makes AQOSA faster to find the optimal point.
In result, this operator is useful for CPU utilization. However, it makes cost and reliability worse. The related result is shown in appendix, section 8.1.

4.2. CPU Change for Performance

Similar to Component Movement operator, the purpose of this operator is reducing the CPU utilization. The experiments are carried out same as the previous operator. Two box plots are created in order to compare the result by AQOSA with and without the operator. Figure 19 shows the box plots.

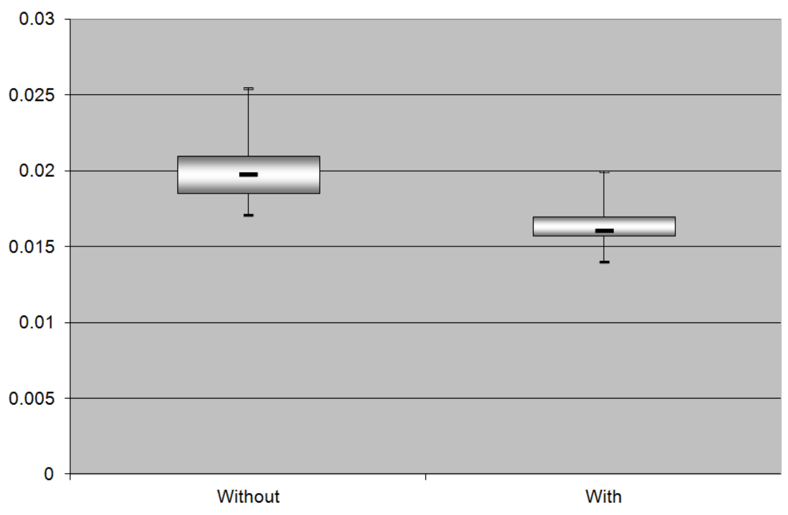


Figure 19: CPU Utilization Box Plot for CPU Change Performance Operator

As it is shown in the figure, CPU utilization is reduced. The index values are presented in the following table.

	Without	With
Minimum	0.017091396	0.013982998
First Quartile	0.018543671	0.015722264
Median	0.019827031	0.016128297
Third Quartile	0.020975818	0.016972927
Maximum	0.025484816	0.019987861

Table 5: CPU Utilization Indexes for CPU Change Performance Operator

Now, a t-test is needed to demonstrate if the decrease is significant or not. The value of "t" is calculated based on the data in table 22 (appendix, section 8.2) as the following:

$$\overline{X}1 = 0.020053508,\ \overline{X}2 = 0.016456135$$

$$SS1 = 0.004080503 - \frac{0.040214316}{10} = 5.90717\text{E-}05$$

$$SS2 = 0.002736328 - \frac{0.027080438}{10} = 2.8284\text{E-}05$$

$$t = \frac{0.020053508 - 0.016456135}{\sqrt{\left[\frac{5.90717\text{E-}05 + 2.8284\text{E-}05}{10^2 - 10}\right]}} = 3.651412873$$

$$Df = (2 \times 10) - 2 = 18$$

Looking at *t* score table for finding the critical *t* value with df=18, it is seen that the calculated "t" value is greater than the critical *t* value at 0.002 probability level:

$$3.651412873 > 3.610$$

Therefore, it is deduced that there is a significant decrease after adding the operator at the 0.002 probability level which is a great confidence level.

The values for iteration are recorded and two box plots are made for them. Figure 20 and table 6 show the result.

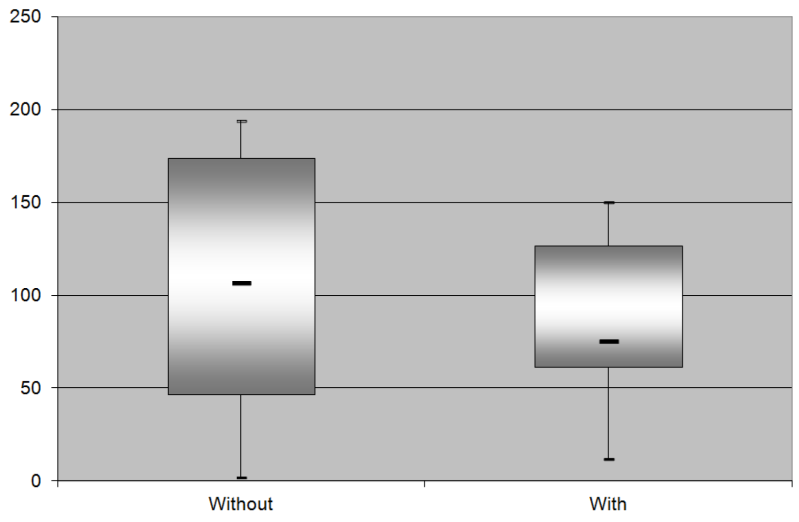


Figure 20: Iteration Box Plot for CPU Change Performance Operator

	Without	With
Minimum	2	12
First Quartile	46.5	61.5
Median	107	75.5
Third Quartile	173.75	126.5
Maximum	194	150

Table 6: Iteration Indexes for CPU Change Performance Operator

As it is seen in the figure, the values of maximum, third quartile, and median are decreased. However, the values of minimum and first quartile are not decreased. Now, a t-test is required to demonstrate whether the decrease of those three indexes makes a significant difference, and subsequently makes a significant reduction or not. Similar to the previous calculation, the "t" value is calculated for iteration based on the data in table 23 (appendix, section 8.2). The result is 0.731679139 which is greater than the critical *t* value got from *t* score table at 0.5 probability level:

$$0.731679139 > 0.688$$

It is deduced that there is a significant decrease at 0.5 probability level. Thus, the operator makes AQOSA faster to find the optimal point.

In result, this operator is useful for CPU utilization. Nevertheless, it makes cost and reliability worse which are shown in appendix, section 8.2.

4.3. CPU Change for Cost

The aim of this operator is to decrease cost. The experiments are carried out same as the previous operators. Since cost is the concerned value, 10 values for cost are recorded for the cases of AQOSA with and without the operator. They are presented in table 26 (appendix, section 8.3). The value of cost that AQOSA offers is the cost for the whole system divided to the maximum cost – decided by the architect of the system – which is stated in the input model. Figure 21 and table 7 present the result. As it is seen in the picture and by comparing the index values, it is deduced that cost is decreased after adding the operator.

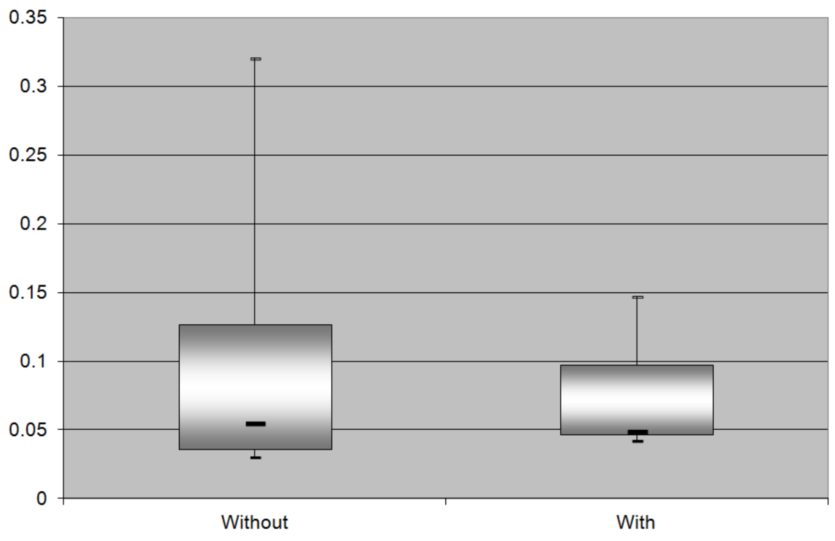


Figure 21: Cost Box Plot for CPU Change Cost Operator

	Without	With
Minimum	0.03	0.042
First Quartile	0.03625	0.046375
Median	0.05475	0.049
Third Quartile	0.126875	0.097125
Maximum	0.3205	0.147

Table 7: Cost Indexes for CPU Change Cost Operator

A t-test is required to see whether the decrease is significant or not. The value of "t" is calculated based on the data in table 26 (appendix, section 8.3) as the following:

$$\overline{X}1 = 0.10045,\ \overline{X}2 = 0.0736$$

$$SS1 = 0.18036175 - \frac{1.00902025}{10} = 0.079459725$$

$$SS2 = 0.0694565 - \frac{0.541696}{10} = 0.0152869$$

$$t = \frac{0.10045 - 0.0736}{\sqrt{\left[\frac{0.079459725 + 0.0152869}{10^2 - 10}\right]}} = 0.827529854$$

$$Df = (2 \times 10) - 2 = 18$$

Looking at *t* score table for finding the critical *t* value with df=18, it is seen that the calculated "t" value is greater than the critical *t* value at 0.5 probability level:

$$0.827529854 > 0.688$$

Therefore, it is concluded that there is a significant decrease after adding the operator at 0.5 probability level.

Subsequently, iteration values are recorded. Figure 22 and table 8 show the distribution of the numbers is decreased.

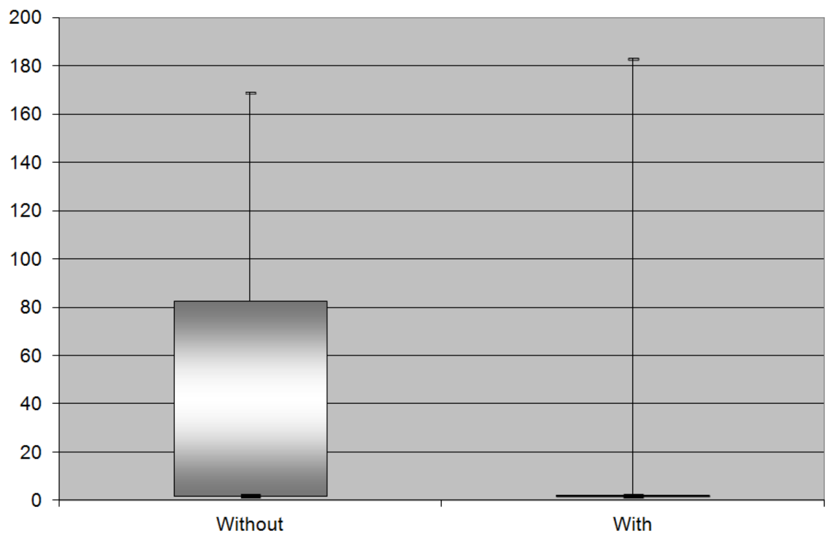


Figure 22: Iteration Box Plot for CPU Change Cost Operator

	Without	With
Minimum	2	2
First Quartile	2	2
Median	2	2
Third Quartile	82.75	2
Maximum	169	183

Table 8: Iteration Indexes for CPU Change Cost Operator

However, a t-test is required to see whether the decrease is significant or not. Same as the previous calculation, the "t" value is calculated for iteration based on the data in table 27 (appendix, section 8.3). The result is 0.861585597 which is greater than the critical *t* value got from *t* score table at 0.5 probability level:

$$0.861585597 > 0.688$$

Thus, there is a significant decrease the 0.5 probability level. Consequently, the operator makes AQOSA faster to find the optimal point.
In result, the operator is efficient for cost. Nevertheless, it has no effect on CPU utilization, and makes reliability worse. They are presented in appendix, section 8.3.

4.4. CPU Change for Reliability

The purpose of this operator is reducing the failure probability and consequently increasing reliability. The failure probability for the whole system is offered by AQOSA output. Same as the previous operators, this value is recorded for 10 times in two cases of AQOSA with and without the operator. They are presented in table 30 (appendix, section 8.4). Two box plots are made to compare the result. Figure 23 and table 9 show the result of box plots for failure probability.

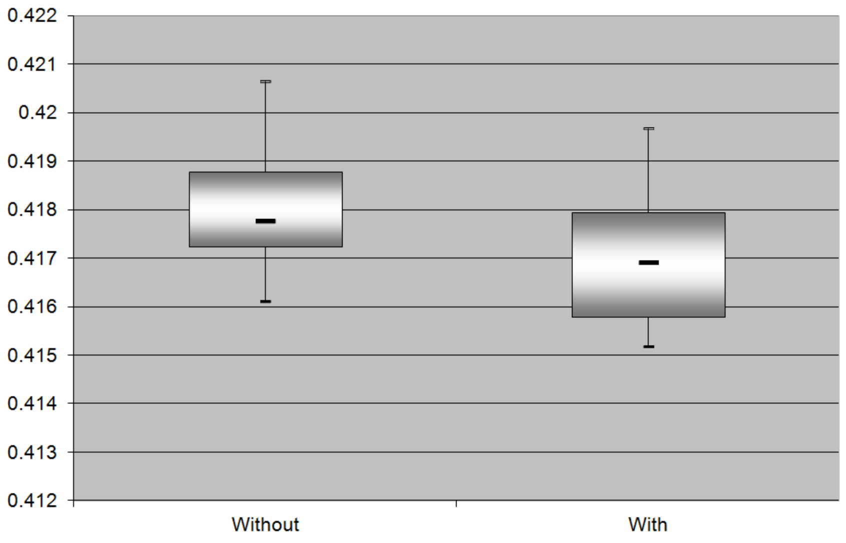


Figure 23: Failure Probability Box Plot for CPU Change Reliability Operator

	Without	With
Minimum	0.416118734	0.41446903
First Quartile	0.417233546	0.415645747
Median	0.417775123	0.416569958
Third Quartile	0.418779501	0.417151746
Maximum	0.420649423	0.419342218

Table 9: Failure Probability Indexes for CPU Change Reliability Operator

As it is seen in the picture and by comparing the indexes, it is deduced that the failure probability is reduced.

To determine whether the reduction is significant or not, a t-test is required. The value of "t" is calculated based on the data in table 30 (appendix, section 8.4) as the following:

$$\overline{X}1 = 0.418013816,\ \overline{X}2 = 0.416690475$$

$$SS1 = 1.747373679 - \frac{17.47355506}{10} = 1.81728\text{E-}05$$

$$SS2 = 1.736329532 - \frac{17.36309522}{10} = 2.00103\text{E-}05$$

$$t = \frac{0.418013816 - 0.416690475}{\sqrt{\left[\frac{1.81728E-05 + 2.00103E-05}{10^2 - 10}\right]}} = 2.031689675$$

$$Df = (2 \times 10) - 2 = 18$$

Looking at *t* score table for finding the critical *t* value with df=18, it is seen that the calculated "t" value is greater than the critical *t* value at 0.1 probability level:

$$2.031689675 > 1.734$$

Therefore, it is deduced that there is a significant decrease for failure probability, and consequently a significant increase for reliability after adding the operator at 0.1 probability level.

The values for iteration are recorded for both cases, and the result is presented in two box plots to compare. Figure 24 and table 10 show the result.

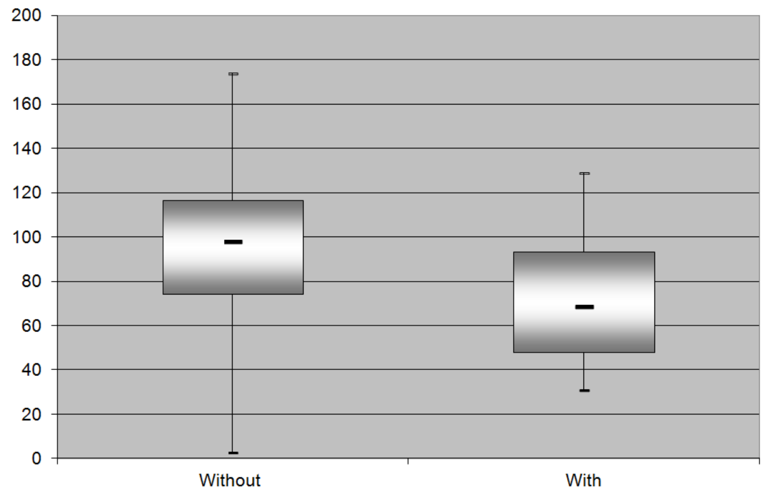


Figure 24: Iteration Box Plot for CPU Change Reliability Operator

	Without	With
Minimum	3	31
First Quartile	74.25	48
Median	98	68.5
Third Quartile	116.5	93.5
Maximum	174	129

Table 10: Iteration Indexes for CPU Change Reliability Operator

As it is seen, the values of maximum, third quartile, median and first quartile are decreased. However, the value of minimum is not decreased. Now, a t-test is required to demonstrate whether the decrease of those four indexes makes a significant difference, and subsequently makes a significant reduction or not. Same as the previous calculation, the "t" value is calculated for iteration based on the data in table 31 (appendix, section 8.4). The result is 0.962678833 which is greater than the critical *t* value got from *t* score table at 0.4 probability level:

$$0.962678833 > 0.861$$

It is deduced that there is a significant decrease at 0.4 probability level. Thus, the operator makes AQOSA faster to find the optimal point.

In result, the operator is efficient for reliability as it decreases failure probability. Nevertheless, it makes CPU utilization worse, and has no considerable effect on cost. They are presented in appendix, section 8.4.

4.5. Combined Operators

As it was shown in the previous sections, each operator is useful for the relevant quality attribute that is made for. However, it might have no effect on the other quality properties, and in some cases it deteriorates them. They have been demonstrated in the appendix. For example, “Component Movement” and “CPU Change for Performance” operators are efficient to decrease CPU Utilization while they are not useful for cost and failure probability as they increase them. “CPU Change for Cost” is efficient to decrease cost while it has no effect on CPU Utilization and increases failure probability. “CPU Change for Reliability” increases reliability, nevertheless; it makes CPU Utilization worse, and has no considerable effect on cost.

When the operators are combined to work together in the framework, each operator acts toward deriving the better objective. Since the optimization process of the framework is based on Genetic Algorithms which optimizes multiple objectives in different directions at the same time, when the operators are set in the framework, the result will be optimized for all quality properties.

According to the description of flow of the work in section 3.2, two parents are needed to be operated by the genetic operators and generate two offsprings. Afterward, the offsprings are studied by the operators for any bottleneck. The operators can be called sequentially or randomly to work on offsprings. The following cases are considered for calling operators to work on every pair of offspring:

1. Randomly for both offsprings
2. Sequentially for both offsprings
3. Randomly for one offspring, and Sequentially for the other one
4. Randomly for one offspring, and no operator for the other one
5. Sequentially for one offspring, and no operator for the other one
6. Half randomly and half sequentially for one offspring, and no operator for the other one

The above cases are considered to examine in order to find the best way of calling operators in the optimization process of the framework. When they are called sequentially, the number of times that they are called will be equal for all of them in the optimization process. When they are called randomly, this number will not be equal. It is important to study the result of the operators when they affect offsprings equally or not equally. Also, it is important to see the result when both offsprings are affected by the operators or just one of them is affected. The following figures show the box plots for CPU utilization, cost and failure probability respectively in 6 stated cases, and also AQOSA without the operators.

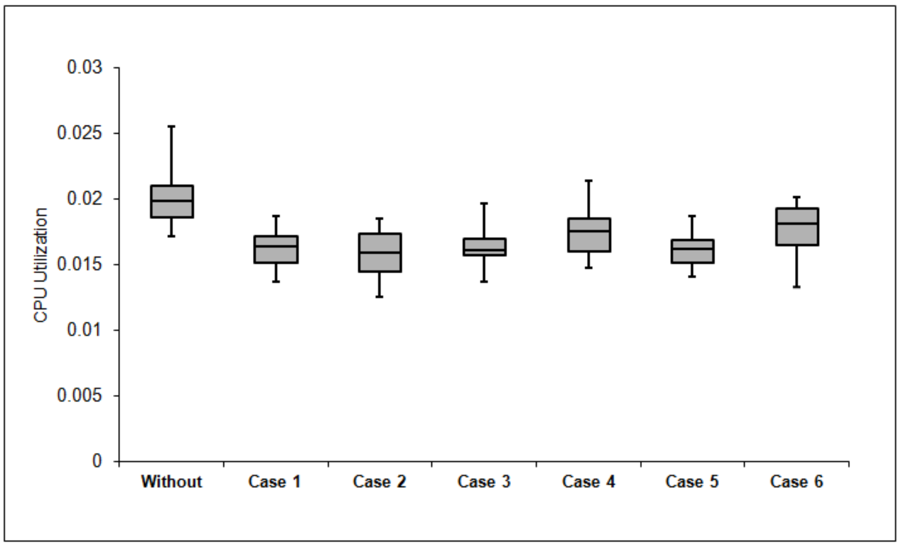


Figure 25: CPU Utilization Box Plot for Combined Operators

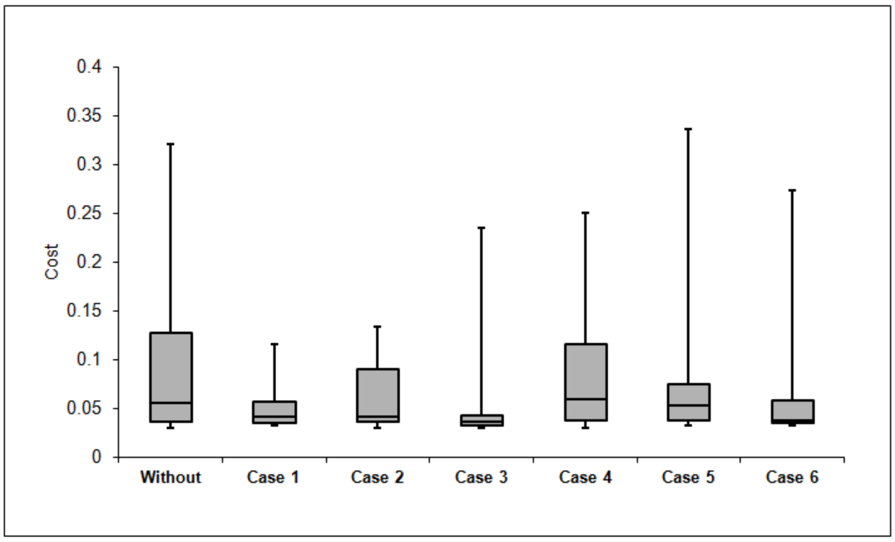


Figure 26: Cost Box Plot for Combined Operators

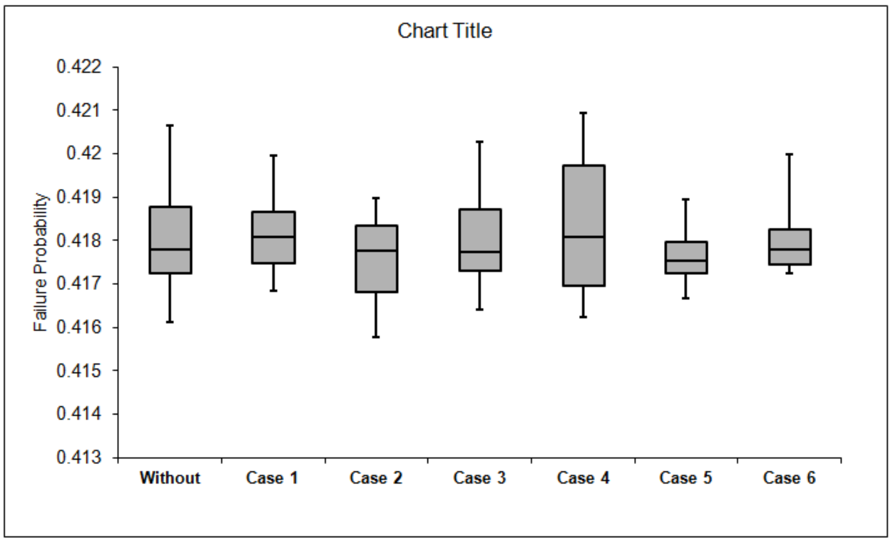


Figure 27: Failure Probability Box Plot for Combined Operators

The related index values are presented in the following tables.

	Without	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6
Min	0.02548482	0.01863501	0.01851739	0.01963184	0.02131908	0.01865552	0.0200867
1’st Q.	0.02097582	0.01710434	0.01734416	0.0169354	0.01849081	0.01684622	0.01923782
Med.	0.01982703	0.01633323	0.01589642	0.01608281	0.01751399	0.01614905	0.01812103
3’rd Q.	0.01854367	0.01512138	0.01445697	0.01566623	0.01598152	0.01506491	0.01646889
Max	0.0170914	0.01363434	0.01252715	0.01364156	0.01468433	0.01406992	0.01329418

Table 11: CPU Utilization Indexes for Combined Operators

	Without	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6
Min	0.3205	0.115	0.133	0.2345	0.2505	0.336	0.2735
1’st Q.	0.126875	0.05625	0.089375	0.041875	0.11575	0.074	0.05725
Med.	0.05475	0.04125	0.04125	0.03625	0.05925	0.0525	0.0375
3’rd Q.	0.03625	0.035	0.035625	0.0325	0.0375	0.0375	0.035
Max	0.03	0.0325	0.03	0.03	0.03	0.0325	0.0325

Table 12: Cost Indexes for Combined Operators

	Without	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6
Min	0.42064942	0.41994633	0.41896813	0.42026856	0.42094918	0.41895853	0.41998823
1’st Q.	0.4187795	0.41864582	0.41833422	0.41871849	0.41971155	0.41796619	0.4182519
Med.	0.41777512	0.4180897	0.4177706	0.41772293	0.41807186	0.41751412	0.41778211
3’rd Q.	0.41723355	0.41745893	0.41679593	0.41728403	0.41693976	0.41724834	0.41745419
Max	0.41611873	0.41683054	0.41575869	0.41640074	0.41622622	0.41667898	0.41723186

Table 13: Failure Probability Indexes for Combined Operators

Although some boxes show decrease or increase clearly, it is needed to do a t-test for all of them in order to find the significance level of the difference, and subsequently concluding a best way to call the operators. The following table presents the “t” value calculated based on the data presented in tables 35, 36 and 37 (appendix, section 8.6).

Property / Case	CPU Utilization	Cost	Failure Probability
1	4.197457318	1.637852483	-0.24837881
2	4.238168343	1.180892917	0.721988629
3	4.009131416	1.1679657	-0.176185749
4	2.302606966	0.144226865	-0.526441421
5	4.31744444	-0.019547609	0.821027578
6	2.224647213	0.649182327	-0.024620831

Table 14: Calculated “t” value for Combined Operators

As it is seen, the "t" value is negative in some cases. The negative value is the result of subtracting mean value of the data set for AQOSA without the operators from the mean value of the data set for AQOSA with the operators, which implies there is an increase after adding the operators. So, the cases with negative "t" are removed. As it is seen, case 1, 3, 4 and 6 represent an increase for failure probability, so they are omitted. Case 5 shows an increase for cost which is removed as well. Consequently, case 2 is selected. Nevertheless, the calculated "t" values should be compared with the critical *t* value got from *t* score table to see if the decrease is significant or not.
Comparing CPU utilization "t" value with the critical *t* value in *t* table shows that there is a significant decrease at 0.001 probability level which is quite confident.

$$4.238168343 > 3.922$$

Comparing cost "t" value with the critical *t* value in *t* table shows that there is a significant decrease at 0.1 probability level.

$$1.180892917 > 1.734$$

Comparing failure probability "t" value with the critical *t* value in *t* table shows that there is a significant decrease at 0.5 probability level.

$$0.721988629 > 0.688$$

In result, the operators are chosen to be called sequentially.
The iterations of the related quality attributes for the selected case are presented in the following box plots and tables.

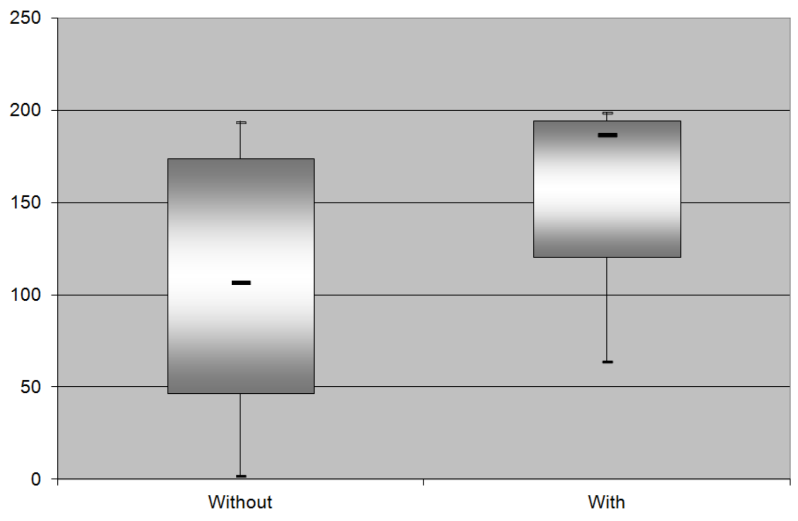


Figure 28: Iteration Related to CPU Utilization for Combined Operators

	Without	With
Minimum	2	64
First Quartile	46.5	120.5
Median	107	187
Third Quartile	173.75	194.25
Maximum	194	199

Table 15: Iteration Indexes Related to CPU Utilization for Combined Operators

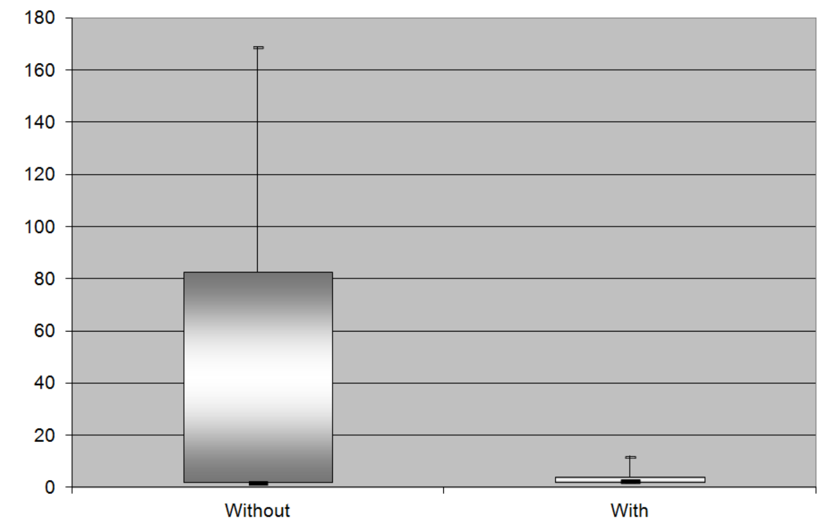


Figure 29: Iteration Related to Cost for Combined Operators

	Without	With
Minimum	2	2
First Quartile	2	2
Median	2	2.5
Third Quartile	82.75	4
Maximum	169	12

Table 16: Iteration Indexes Related to Cost for Combined Operators

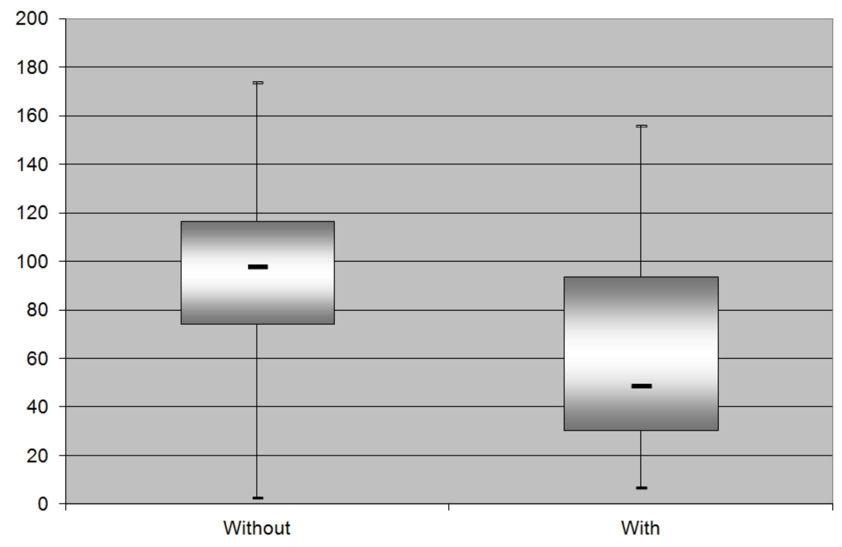


Figure 30: Iteration Related to Failure Probability for Combined Operators

	Without	With
Minimum	3	7
First Quartile	74.25	30.5
Median	98	49
Third Quartile	116.5	93.75
Maximum	174	156

Table 17: Iteration Indexes Related to Failure Probability for Combined Operators

As it is seen in the box plots, and also by comparing index values, it is deduced that there is an increase for the iteration of CPU utilization while there are decreases for the iterations of cost and failure probability. So, when the operators are combined there is no good point for finding the optimal point of CPU utilization faster. For the other two iterations, there is a need to do t-test in order to see whether the decrease is significant or not. It is calculated similar to the previous ones based on the data presented in table 38 and 39 (appendix, section 8.6). Looking at the *t* score table and comparing the critical *t* value to the calculated "t" value demonstrates the following result for the iteration of cost and failure probability respectively:

- 1.895459528 > 1.734 at 0.1 probability level
- 1.14309447 > 0.688 at 0.5 probability level

Therefore, there is a significant decrease for the iteration of cost at 0.1 probability level; also there is a significant decrease for the iteration of failure probability at 0.5 probability level. Consequently, the combined operators help AQOSA to find the optimal point of cost and failure probability faster.

5. Discussion

Diagnosing and healing bottlenecks in the early phases of developing a system promises to produce a successful system. In the view of this, the aim of the research was formed to make the job automated. The theory was practiced in an existing framework which generates optimized architecture designs automatically. To serve this purpose, some antipatterns were studied. Subsequently, some operators were designed and developed based on the antipatterns. The result demonstrated the operators are efficient for the purpose.

However, it was not possible to implement all antipatterns due to the restrictions of the framework which were introduced in section 3.3.

Threshold was introduced as a fundamental criterion for every operator. Since it needs to be changed for different models due to different resources, it should be defined for every study model before the process of optimization. One of the difficulties of this research was finding a right threshold which was done by try and error. As it was stated in section 3.4.1, it should be determined for every single operator firstly. When the operators are combined, they may not be as effective as they were in the single operators. Hence, for every single operator two or three options are selected. Then they should be examined again after combining the operators in order to find the right one. Since there was no rule to find how the threshold affects the result, all this process was done by try and error which was time consuming.

6. Conclusion

This research was done upon a previous research by Etemadi R. et al. [10]. The previous work offers AQOSA framework which generates optimized architecture designs automatically. It considers five quality attributes for optimization including CPU utilization, bus utilization, response time, cost and reliability. The aim of this research was extending the framework in that the result is lack of any bottleneck for three quality attributes of CPU utilization, cost and reliability. To serve this purpose, I designed and implemented some operators which diagnose and heal bottlenecks automatically. Subsequently, I carried out some types of experiments in order to validate the result. The result demonstrated each operator is useful individually. Moreover, the combined operators work quite efficiently within AQOSA.

In result, AQOSA framework is now improved by the intelligent operators, and generates optimized architecture designs which do not have bottlenecks for the stated quality attributes, and assures to develop a successful system.

As it was stated in the previous section, finding a right threshold is done by try and error which is time consuming. Also, when the operators are combined, the found threshold may not be as effective as it is in the single operator. The reason is still undiscovered. Hence, the future work of this research could be finding the reason, also a better way to find the right threshold.

7. References

[1] Alander J. T., An Indexed Bibliography of Genetic Algorithms in Testing, Univ. of Vaasa, Finland, Tech. Rep. 94-1-TEST, 2008.
[2] Aleti A., Software Architecture Optimization Methods: A Systematic Literature Review, IEEE, 2012.
[3] Back T, Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms, Oxford University Press, 1996.
[4] Bass L. et al., Software Architecture in Practice, Addison Wesley, 2003.
[5] Bajpai P. et al., Genetic Algorithm – an Approach to Solve Global Optimization Problems, Indian Journal of Computer Science and Engineering, vol. 1, No 3, pp. 199-206, 2010.
[6] Brown W. J. et al., AntiPatterns: Refactoring Software, Architecture, and Project in Crisis, 1998.
[7] Blum C. et al., Metahuristics in Combinatorial Optimization: Overview and conceptual comparison, ACM Computing Surveys 35, pp. 268-308, 2003.
[8] Chaudron M. et al., Towards Automated Software Architectures Design Using Model Transformations and Evolutionary Algorithms, GECCO (Companion), ACM, pp. 2097–2098, 2010.
[9] Chaudron M. et al., A Process for Resolving Performance Trade-Offs in Component-Based Architectures, Component-Based Software Engineering, ser. LNCS, vol. 4063, pp. 254–269, 2006.
[10] Etemadi R. et al., An Evolutionary Multiobjective Approach to Component-Based Software Architecture Design, IEEE Congress on Evolutionary Computation, 2011.
[11] Goldberg, D., Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley, 1989.
[12] Grunske L. et al., An Outline of an Architecture-Based Method for Optimizing Dependability attributes of Software-Intensive Systems, Architecture Dependable Systems, ser. Lecture Notes in Computer Science, vol. 4615, pp. 188-209, 2006.
[13] Harman M. et al., The Current State and Future of Search Based Software Engineering, International Conference of Software Engineering, pp. 342-355, 2007.
[14] Jiang H. et al., A Foundational Study on the Applicability of Genetic Algorithm to Software Engineering Problems, Proc. IEEE Congress on Evolutionary Computation (CEC' 07) CPS/IEEE Computer Society, pp. 2210-2219, 2007.
[15] Kremelberg, D., Practical Statistics: A Quick and Easy Guide to SPSS, Stata, and other Statistical Software, SAGE publications, 2009.
[16] Kue W. et al., Recent Advances in Optimal Reliability Allocation, Computational Intelligence in Reliability Engineering, Evolutionary Techniques in Reliability Analysis and Optimization, pp. 1-36, 2007.
[17] Maier M. et al, Software Architecture: Introducing IEEE Standard 1471, Computer, vol. 34, pp. 107-109, 2001.
[18] Neilson J.E. et al., Software Bottlenecking in Client Server Systems and Rendezvous Networks, IEEE, vol. 21, pp. 776-782, 1995.
[19] Oracle white paper, Rapid Bottleneck Identification-A better way to do load testing, 2010.
[20] Prebys, E. K., The Genetic Algorithm in Computer Science, MIT Undergraduate Journal of Mathematics, 2007.

[21] Trubiani C., Detection and Solution of Software Performance Antipatterns in Palladio Architectural Models, ACM, 2011.
[22] Trubiani C., Automated Generation of Architectural Feedback from Software Performance Analysis Results, 2011.
[23] Villegas N. M. et al., A Framework for Evaluating Quality-Driven Self-Adaptive Software Systems, SEAMS, pp. 80-89, 2011.
[24] Williamson D. F. et al., The Box Plot: A Simple Visual Method to Interpret Data, Academic and Clinic, vol. 11, pp. 916-921, 1989.
[25] Woodside M., Tutorial Introduction to Layered Modeling of Software Performance, Carlton University, 2002. 24-25

8. Appendix

8.1. Component Movement

The CPU utilization result of two experiments for AQOSA with and without the operator is presented in table 18.

Without	With
0.017173451	0.022338673
0.0211913	0.014026414
0.017091396	0.018464169
0.022507157	0.019301473
0.01853558	0.022981467
0.025484816	0.015641766
0.020326584	0.018859757
0.018567944	0.018418745
0.02032937	0.016010999
0.019327478	0.021498712

Table 18: Data Sets of CPU Utilization for Component Movement Operator

The iteration result (related to CPU utilization) of two experiments for AQOSA with and without the operator is presented in table 19.

Without	With
84	143
41	95
175	19
2	6
170	2
194	107
16	18
130	23
182	9
63	81

Table 19: Data Sets of Iteration Related to CPU Utilization for Component Movement Operator

Figure 31 shows an increase for cost by the operator. Thus, it makes the cost worse.

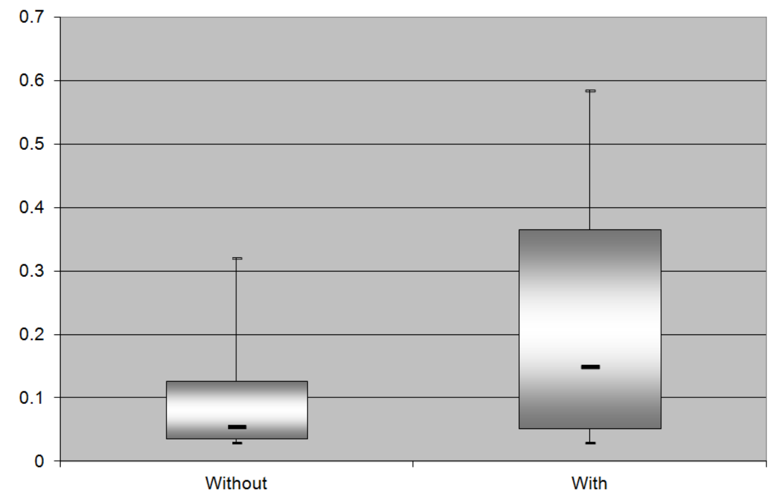


Figure 31: Cost Box Plot for Component Movement Operator

	Without	With
Minimum	0.03	0.03
First Quartile	0.03625	0.0525
Median	0.05475	0.14975
Third Quartile	0.126875	0.365125
Maximum	0.3205	0.585

Table 20: Cost Indexes for Component Movement Operator

Figure 32 shows an increase for failure probability by the operator. So, it makes the reliability worse.

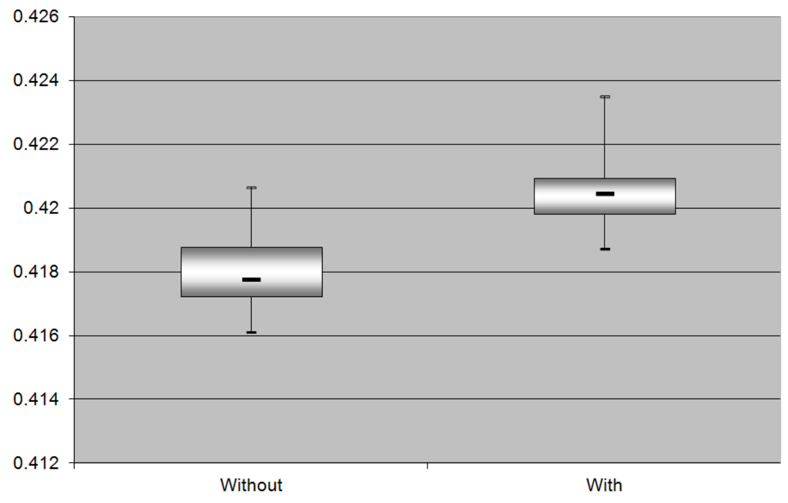


Figure 32: Failure Probability Box Plot for Component Movement Operator

	Without	With
Minimum	0.03	0.03
First Quartile	0.03625	0.0525
Median	0.05475	0.14975
Third Quartile	0.126875	0.365125
Maximum	0.3205	0.585

Table 21: Failure Probability Indexes for Component Movement Operator

8.2. CPU Change for Performance

The CPU utilization result of two experiments for AQOSA with and without the operator is presented in table 22.

Without	With
0.017173451	0.016799698
0.0211913	0.016116486
0.017091396	0.016140107
0.022507157	0.018452441
0.01853558	0.014417244
0.025484816	0.015627605
0.020326584	0.019987861
0.018567944	0.01703067
0.02032937	0.013982998
0.019327478	0.01600624

Table 22: Data Sets of CPU Utilization for CPU Change Performance Operator

The iteration result (related to CPU utilization) of two experiments for AQOSA with and without the operator is presented in table 23.

Without	With
84	150
41	76
175	43
2	133
170	72
194	107
16	12
130	75
182	133
63	58

Table 23: Data Sets of Iteration Related to CPU Utilization for CPU Change Performance Operator

The result of the operator for cost is shown in figure 33. Since the distribution of the numbers is increased by the operator, it makes the cost worse.

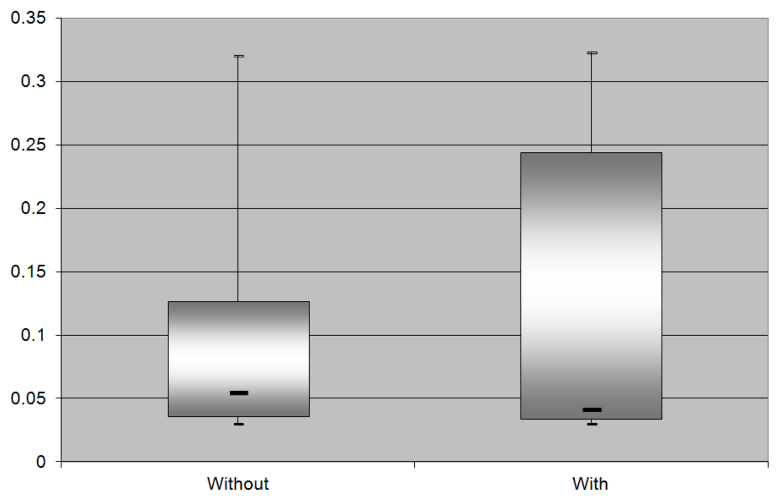


Figure 33: Cost Box Plot for CPU Change Performance Operator

	Without	With
Minimum	0.03	0.03
First Quartile	0.03625	0.03375
Median	0.05475	0.04125
Third Quartile	0.126875	0.244125
Maximum	0.3205	0.323

Table 24: Cost Indexes for CPU Change for Performance Operator

Also, it increases failure probability as it is shown in figure 34. Thus, it makes reliability worse.

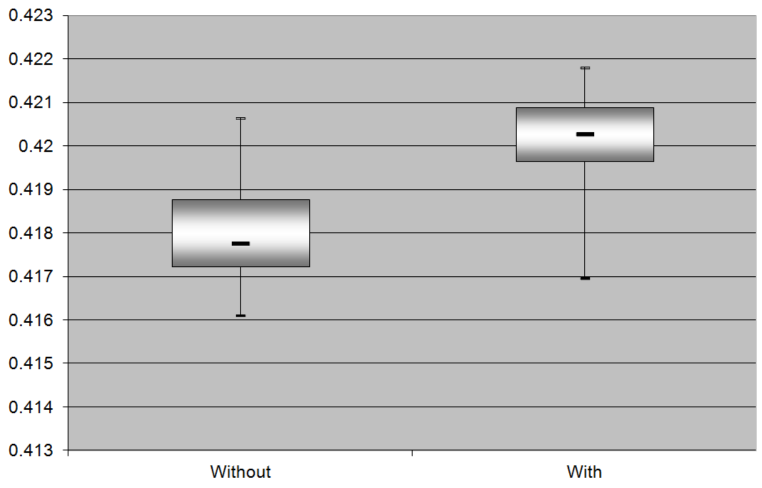


Figure 34: Failure Probability Box Plot for CPU Change Performance Operator

	Without	With
Minimum	0.416118734	0.416973127
First Quartile	0.417233546	0.419644739
Median	0.417775123	0.420287833
Third Quartile	0.418779501	0.420880655
Maximum	0.420649423	0.421815245

Table 25: Failure Probability Indexes for CPU Change Performance Operator

8.3. CPU Change for Cost

The cost result of two experiments for AQOSA with and without the operator is presented in table 26.

Without	With
0.03	0.138
0.1275	0.147
0.035	0.0455
0.04	0.042
0.03	0.049
0.064	0.049
0.0455	0.049
0.3205	0.063
0.187	0.045
0.125	0.1085

Table 26: Data Sets of Cost for CPU Change Cost Operator

The iteration result (related to cost) of two experiments for AQOSA with and without the operator is presented in table 27.

Without	With
2	2
2	183
2	2
2	2
7	2
2	2
2	2
108	2
169	2
149	2

Table 27: Data Sets of Iteration Related to Cost for CPU Change Cost Operator

The result of the operator for CPU utilization is shown in figure 35. As it is seen, the index values of the box plot are approximately equal, which means the operator has no effect for CPU utilization.

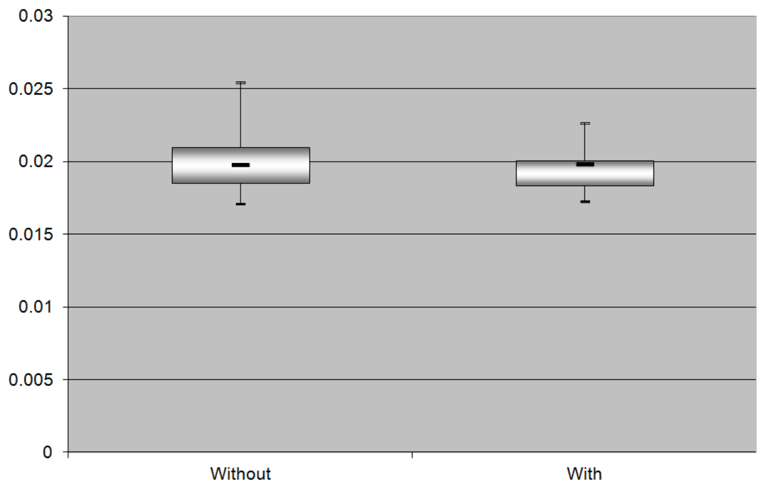


Figure 35: CPU Utilization Box Plot for CPU Change Cost Operator

	Without	With
Minimum	0.017091396	0.017273788
First Quartile	0.018543671	0.018337329
Median	0.019827031	0.019876781
Third Quartile	0.020975818	0.020049067
Maximum	0.025484816	0.022688478

Table 28: CPU Utilization Indexes for CPU Change Cost Operator

The result of the operator for failure probability is shown in figure 36. By comparing the index values before and after adding the operator, it is concluded that the operator is not good for reliability.

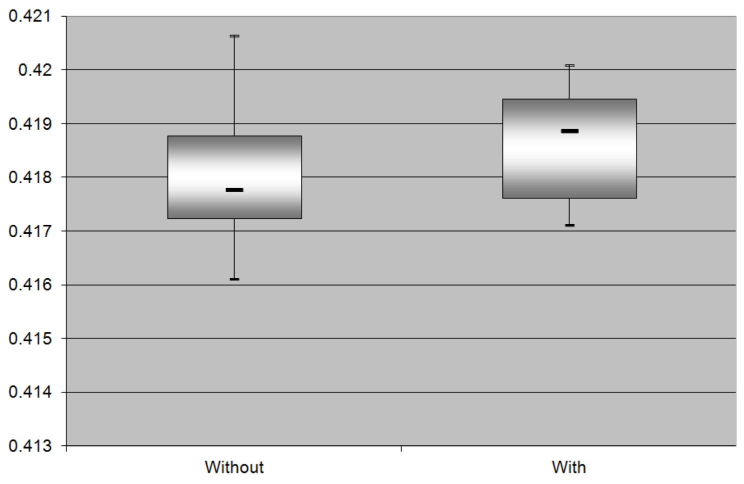


Figure 36: Failure Probability Box Plot for CPU Change Cost Operator

	Without	With
Minimum	0.416118734	0.417121075
First Quartile	0.417233546	0.417611525
Median	0.417775123	0.41886799
Third Quartile	0.418779501	0.419462091
Maximum	0.420649423	0.420101182

Table 29: Failure Probability Indexes for CPU Change Cost Operator

8.4. CPU Change for Reliability

The cost result of two experiments for AQOSA with and without the operator is presented in table 30.

Without	With
0.416118734	0.418769719
0.417834696	0.416367949
0.418886576	0.415887501
0.417289557	0.416995402
0.416343555	0.41720386
0.420649423	0.415565163
0.418458278	0.416771968
0.419626919	0.415531943
0.417715549	0.419342218
0.417214875	0.41446903

Table 30: Data Sets of Failure Probability for CPU Change Reliability Operator

The iteration result (related to cost) of two experiments for AQOSA with and without the operator is presented in table 31.

Without	With
97	63
99	128
174	129
144	31
27	67
3	43
106	70
120	99
90	77
69	34

Table 31: Data Sets of Iteration Related to Failure Probability for CPU Change Reliability Operator

The result of the operator for CPU utilization is shown in figure 37. Since the distribution of the numbers is increased by the operator, it is concluded that it worsens CPU utilization.

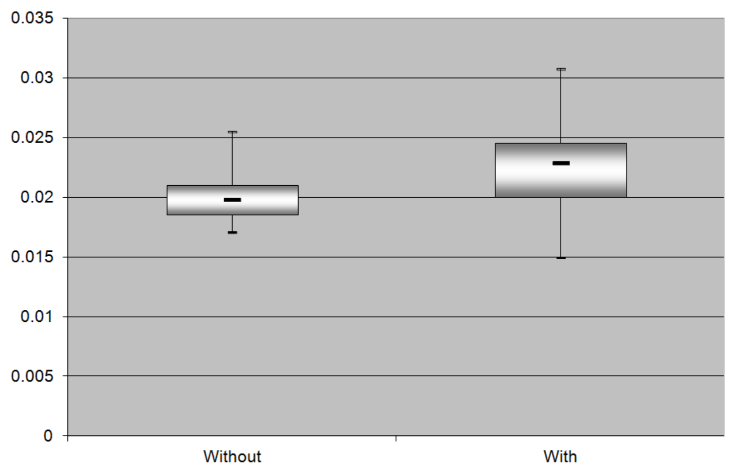


Figure 37: CPU Utilization Box Plot for CPU Change for Reliability Operator

	Without	With
Minimum	0.017091396	0.014982877
First Quartile	0.018543671	0.020039156
Median	0.019827031	0.022890828
Third Quartile	0.020975818	0.024509649
Maximum	0.025484816	0.030753465

Table 32: CPU Utilization Indexes for CPU Change Reliability Operator

The result of the operator for cost is shown in figure 38. Since minimum, first quartile and median values are approximately equal, and maximum value is increased while just third quartile is decreased, it is concluded that the operator has no considerable effect for cost.

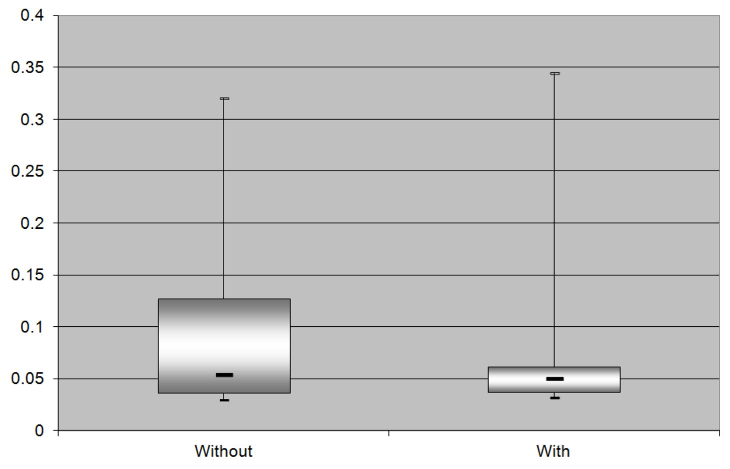


Figure 38: Cost Box Plot for CPU Change for Reliability Operator

	Without	With
Minimum	0.03	0.0325
First Quartile	0.03625	0.0375
Median	0.05475	0.05075
Third Quartile	0.126875	0.061625
Maximum	0.3205	0.3445

Table 33: Cost Indexes for CPU Change Reliability Operator

8.5. Load Balancer

This operator was designed to decrease response time. Since, it was not useful and worsened response time; it was not added to the implementation. The following box plot and table shows the result.

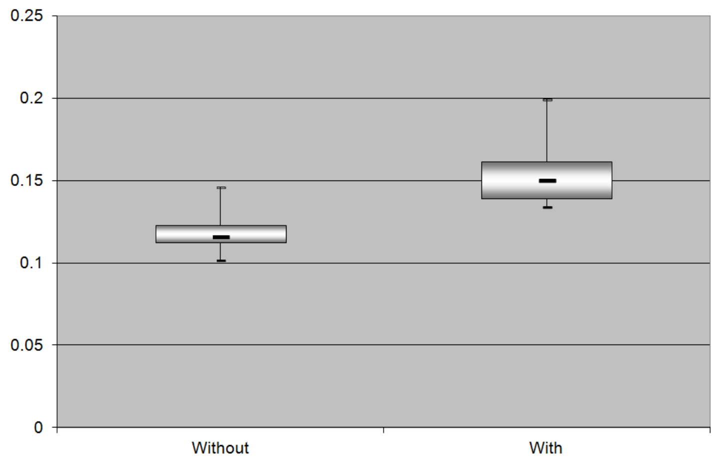


Figure 39: Response Time Box Plot for Load Balancer Operator

	Without	With
Minimum	0.101704056	0.133909116
First Quartile	0.112464857	0.13932213
Median	0.115678347	0.150101236
Third Quartile	0.12296796	0.161686674
Maximum	0.145814313	0.199297148

Table 34: Response Time Indexes for Load Balancer Operator

8.6. Combined Operators

The CPU utilization result of 6 cases for combined operators is presented in table 35.

Case 1	Case 2	Case 3	Case 4	Case 5	Case 6
0.015562	0.018517	0.016967	0.018243	0.015005	0.016338
0.014333	0.0177	0.015676	0.018572	0.016157	0.018824
0.016792	0.014307	0.016094	0.021319	0.017373	0.013294
0.013634	0.013253	0.016072	0.016042	0.017003	0.019899
0.015992	0.014908	0.016959	0.020745	0.015235	0.020087
0.017249	0.015772	0.019632	0.015961	0.018656	0.019332
0.018635	0.016593	0.015663	0.014684	0.01407	0.016428
0.017209	0.012527	0.016866	0.016785	0.015008	0.016592
0.014975	0.017594	0.013642	0.018246	0.016376	0.017418
0.016674	0.016021	0.014818	0.014992	0.016141	0.018956

Table 35: Data Sets of CPU Utilization for 6 Cases of Combined Operators

The cost result of 6 cases for combined operators is presented in table 36.

Case 1	Case 2	Case 3	Case 4	Case 5	Case 6
0.0375	0.035	0.03	0.0735	0.071	0.0375
0.045	0.0875	0.035	0.12	0.336	0.035
0.06	0.0375	0.07	0.214	0.0375	0.0325
0.0325	0.09	0.0425	0.045	0.0675	0.0375
0.045	0.0325	0.04	0.045	0.0375	0.164
0.035	0.045	0.0325	0.103	0.0375	0.2735
0.035	0.133	0.0325	0.035	0.075	0.063
0.063	0.03	0.0325	0.03	0.0375	0.0325
0.115	0.0375	0.0375	0.0325	0.2815	0.04
0.0325	0.1	0.2345	0.2505	0.0325	0.035

Table 36: Data Sets of Cost for 6 Cases of Combined Operators

The failure probability result of 6 cases for combined operators is presented in table 37.

Case 1	Case 2	Case 3	Case 4	Case 5	Case 6
0.416831	0.417387	0.417839	0.416226	0.416679	0.418881
0.418263	0.418363	0.418737	0.418025	0.418031	0.418016
0.417562	0.41653	0.420189	0.419581	0.416954	0.4178
0.418773	0.417561	0.416401	0.416848	0.417193	0.417325
0.418221	0.416599	0.418662	0.419755	0.418018	0.417486
0.419946	0.418715	0.41727	0.420189	0.417514	0.417232
0.419274	0.417981	0.417328	0.416839	0.417514	0.417443
0.417425	0.418968	0.416918	0.420949	0.418959	0.418331
0.417234	0.415759	0.420269	0.418119	0.417812	0.417764
0.417958	0.418247	0.417607	0.417216	0.417414	0.419988

Table 37: Data Sets of Failure Probability for 6 Cases of Combined Operators

The iteration result related to cost for combined operators in case 2 (sequentially) is presented in table 38.

Without	With
2	2
2	4
2	4
2	2
7	2
2	3
2	8
108	2
169	2
149	3

Table 38: Data Sets of Iteration Related to Cost for Combined Operators Sequentially

The iteration result related to failure probability for combined operators in case 2 (sequentially) is presented in table 39.

Without	With
97	63
99	155
174	28
144	7
27	38
3	40
106	156
120	10
90	104
69	58

Table 39: Data Sets of Iteration Related to Failure Probability for Combined Operators Sequentially

8.7. Input Model for AQOSA

```
<?xml version="1.0" encoding="UTF-8"?>
<aqosa.ir:AQOSAModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aqosa.ir="http://se.liacs.nl/aqosa/ir">
  <assembly>
    <component name="ReadWheelSpeedSensors">
      <service name="ReadWheelSpeedSensors"/>
      <inport name="ReadWheelSpeedSensors-in"/>
      <outport name="ReadWheelSpeedSensors-out"/>
    </component>
    <component name="ControlWheelSpeed">
      <service name="CalculateWheelRotation"/>
      <inport name="ControlWheelSpeed-in"/>
      <outport name="ControlWheelSpeed-out"/>
    </component>
    <component name="EngineVehicleInterface">
      <service name="ObtainEngineSpeed"/>
      <service name="ObtainVehicleSpeed"/>
      <service name="ObtainCoolantTemp"/>
      <inport name="EngineVehicleInterface-in_Engine"/>
      <inport name="EngineVehicleInterface-in_Vehicle"/>
      <inport name="EngineVehicleInterface-in_Coolant"/>
      <outport name="EngineVehicleInterface-out_Engine"/>
      <outport name="EngineVehicleInterface-out_Vehicle"/>
      <outport name="EngineVehicleInterface-out_Coolant"/>
    </component>
    <component name="ProvidePowerModeInfo">
      <service name="PowerModeInfo"/>
      <outport name="PowerModeInfo-out"/>
    </component>
    <component name="ControlEngineSpeedGauge">
      <service name="DisplayEngineSpeed"/>
      <inport name="ControlEngineSpeedGauge-in"/>
      <outport name="ControlEngineSpeedGauge-out"/>
    </component>
    <component name="ControlVehicleSpeedGauge">
      <service name="DisplayVehicleSpeed"/>
      <inport name="ControlVehicleSpeedGauge-in"/>
      <outport name="ControlVehicleSpeedGauge-out"/>
    </component>
    <component name="Gauge_Engine">
      <service name="CalculateNeedlePosition"/>
      <inport name="Gauge_Engine-in"/>
    </component>
    <component name="TransmissionVehicleInterface">
      <service name="ReadLeverPstn"/>
      <outport name="TransmissionVehicleInterface-out"/>
    </component>
    <component name="ControlGearSelectedIndication">
      <service name="GearDisplayValue"/>
      <inport name="ControlGearSelectedIndication-in"/>
      <outport name="ControlGearSelectedIndication-out"/>
    </component>
    <component name="Display_Engine">
      <service name="IndicateGearPstn"/>
      <service name="DisplayOAT"/>
      <service name="DisplayOdometer"/>
      <service name="IndicateLowWasher"/>
      <inport name="Display_Engine-in_Gear"/>
      <inport name="Display_Engine-in_OAT"/>
      <inport name="Display_Engine-in_Odometer"/>
      <inport name="Display_Engine-in_Washer"/>
    </component>
    <component name="ReadOATSensor">
      <service name="ObtaionOAT"/>
      <outport name="ReadOATSensor-out"/>
    </component>
    <component name="ControlOutsideAirTemp">
      <service name="CalculateOAT"/>
      <inport name="ControlOutsideAirTemp-in"/>
      <outport name="ControlOutsideAirTemp-out"/>
    </component>
    <component name="ControlCoolantTempGauge">
      <service name="DisplayCoolantTemp"/>
      <inport name="ControlCoolantTempGauge-in"/>
      <outport name="ControlCoolantTempGauge-out"/>
    </component>
    <component name="ReadDriverDoorAjarSwitch">
      <service name="ReadDriverDoorAjarSwitch"/>
      <outport name="ReadDriverDoorAjarSwitch-out"/>
    </component>
    <component name="ControlOdometer">
      <service name="OdometerValue"/>
      <inport name="ControlOdometer-in"/>
      <outport name="ControlOdometer-out"/>
    </component>
    <component name="ReadTripStemButton">
      <service name="ReadTripStemButton"/>
      <outport name="ReadTripStemButton-out"/>
    </component>
    <component name="ReadLowWasherLevel">
      <service name="ReadLowWasherLevel"/>
      <outport name="ReadLowWasherLevel-out"/>
    </component>
    <component name="ControlWasherLevelIndication">
      <service name="ControlWasherLevelIndication"/>
      <inport name="ControlWasherLevelIndication-in"/>
      <outport name="ControlWasherLevelIndication-out"/>
    </component>
    <flow name="Ignition_to_EngineSpeed">
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.3/@service.0"/>
      <action xsi:type="aqosa.ir:CommunicateAction"
source="//@assembly/@component.3/@outport.0"
destination="//@assembly/@component.2/@inport.0"/>
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.2/@service.0"/>
      <action xsi:type="aqosa.ir:CommunicateAction"
source="//@assembly/@component.2/@outport.0"
destination="//@assembly/@component.4/@inport.0"/>
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.4/@service.0"/>
      <action xsi:type="aqosa.ir:CommunicateAction"
source="//@assembly/@component.4/@outport.0"
destination="//@assembly/@component.6/@inport.0"/>
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.6/@service.0"/>
    </flow>
    <flow name="Ignition_to_VehicleSpeed">
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.3/@service.0"/>
      <action xsi:type="aqosa.ir:CommunicateAction"
source="//@assembly/@component.3/@outport.0"
destination="//@assembly/@component.0/@inport.0"/>
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.0/@service.0"/>
      <action xsi:type="aqosa.ir:CommunicateAction"
source="//@assembly/@component.0/@outport.0"
destination="//@assembly/@component.1/@inport.0"/>
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.1/@service.0"/>
      <action xsi:type="aqosa.ir:CommunicateAction"
source="//@assembly/@component.1/@outport.0"
destination="//@assembly/@component.2/@inport.1"/>
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.2/@service.1"/>
      <action xsi:type="aqosa.ir:CommunicateAction"
source="//@assembly/@component.2/@outport.1"
destination="//@assembly/@component.5/@inport.0"/>
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.5/@service.0"/>
      <action xsi:type="aqosa.ir:CommunicateAction"
source="//@assembly/@component.5/@outport.0"
```

```
destination="//@assembly/@component.6/@inport.0"/>
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.6/@service.0"/>
    </flow>
    <flow name="GearIndication">
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.7/@service.0"/>
      <action xsi:type="aqosa.ir:CommunicateAction"
source="//@assembly/@component.7/@outport.0"
destination="//@assembly/@component.8/@inport.0"/>
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.8/@service.0"/>
      <action xsi:type="aqosa.ir:CommunicateAction"
source="//@assembly/@component.8/@outport.0"
destination="//@assembly/@component.9/@inport.0"/>
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.9/@service.0"/>
    </flow>
    <flow name="VehicleSpeedIndication">
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.0/@service.0"/>
      <action xsi:type="aqosa.ir:CommunicateAction"
source="//@assembly/@component.0/@outport.0"
destination="//@assembly/@component.1/@inport.0"/>
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.1/@service.0"/>
      <action xsi:type="aqosa.ir:CommunicateAction"
source="//@assembly/@component.1/@outport.0"
destination="//@assembly/@component.2/@inport.1"/>
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.2/@service.1"/>
      <action xsi:type="aqosa.ir:CommunicateAction"
source="//@assembly/@component.2/@outport.1"
destination="//@assembly/@component.5/@inport.0"/>
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.5/@service.0"/>
      <action xsi:type="aqosa.ir:CommunicateAction"
source="//@assembly/@component.5/@outport.0"
destination="//@assembly/@component.6/@inport.0"/>
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.6/@service.0"/>
    </flow>
    <flow name="EngineSpeedIndication">
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.2/@service.0"/>
      <action xsi:type="aqosa.ir:CommunicateAction"
source="//@assembly/@component.2/@outport.0"
destination="//@assembly/@component.4/@inport.0"/>
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.4/@service.0"/>
      <action xsi:type="aqosa.ir:CommunicateAction"
source="//@assembly/@component.4/@outport.0"
destination="//@assembly/@component.6/@inport.0"/>
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.6/@service.0"/>
    </flow>
    <flow name="OATCalculation">
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.10/@service.0"/>
```

```
      <action xsi:type="aqosa.ir:CommunicateAction"
source="//@assembly/@component.10/@outport.0"
destination="//@assembly/@component.11/@inport.0"/>
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.11/@service.0"/>
      <action xsi:type="aqosa.ir:CommunicateAction"
source="//@assembly/@component.11/@outport.0"
destination="//@assembly/@component.9/@inport.1"/>
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.9/@service.1"/>
    </flow>
    <flow name="EngineCoolantTemp">
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.2/@service.2"/>
      <action xsi:type="aqosa.ir:CommunicateAction"
source="//@assembly/@component.2/@outport.2"
destination="//@assembly/@component.12/@inport.0"/>
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.12/@service.0"/>
      <action xsi:type="aqosa.ir:CommunicateAction"
source="//@assembly/@component.12/@outport.0"
destination="//@assembly/@component.6/@inport.0"/>
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.6/@service.0"/>
    </flow>
    <flow name="DriverDoor_to_Odometer">
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.13/@service.0"/>
      <action xsi:type="aqosa.ir:CommunicateAction"
source="//@assembly/@component.13/@outport.0"
destination="//@assembly/@component.14/@inport.0"/>
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.14/@service.0"/>
      <action xsi:type="aqosa.ir:CommunicateAction"
source="//@assembly/@component.14/@outport.0"
destination="//@assembly/@component.9/@inport.2"/>
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.9/@service.2"/>
    </flow>
    <flow name="StemButton_to_Odometer">
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.15/@service.0"/>
      <action xsi:type="aqosa.ir:CommunicateAction"
source="//@assembly/@component.15/@outport.0"
destination="//@assembly/@component.14/@inport.0"/>
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.14/@service.0"/>
      <action xsi:type="aqosa.ir:CommunicateAction"
source="//@assembly/@component.14/@outport.0"
destination="//@assembly/@component.9/@inport.2"/>
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.9/@service.2"/>
    </flow>
    <flow name="LowWasherIndication">
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.16/@service.0"/>
      <action xsi:type="aqosa.ir:CommunicateAction"
source="//@assembly/@component.16/@outport.0"
destination="//@assembly/@component.17/@inport.0"/>
```

```xml
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.17/@service.0"/>
      <action xsi:type="aqosa.ir:CommunicateAction"
source="//@assembly/@component.17/@outport.0"
destination="//@assembly/@component.9/@inport.3"/>
      <action xsi:type="aqosa.ir:ComputeAction"
service="//@assembly/@component.9/@service.3"/>
    </flow>
  </assembly>
  <scenarios>
    <flowset name="Average" completionTime="5000.0" missedPercentage="0.02">
      <flowinstance instance="//@assembly/@flow.0" start="500.0" trigger="5000.0"
deadline="150.0"/>
      <flowinstance instance="//@assembly/@flow.1" start="500.0" trigger="5000.0"
deadline="150.0"/>
      <flowinstance instance="//@assembly/@flow.2" start="500.0" trigger="5000.0"
deadline="100.0"/>
      <flowinstance instance="//@assembly/@flow.3" start="1000.0" trigger="100.0"
deadline="50.0"/>
      <flowinstance instance="//@assembly/@flow.4" start="1000.0" trigger="100.0"
deadline="50.0"/>
      <flowinstance instance="//@assembly/@flow.5" start="1000.0" trigger="1000.0"
deadline="100.0"/>
      <flowinstance instance="//@assembly/@flow.6" start="1000.0" trigger="100.0"
deadline="50.0"/>
      <flowinstance instance="//@assembly/@flow.7" start="500.0" trigger="5000.0"
deadline="500.0"/>
      <flowinstance instance="//@assembly/@flow.8" start="500.0" trigger="5000.0"
deadline="500.0"/>
      <flowinstance instance="//@assembly/@flow.9" start="500.0" trigger="5000.0"
deadline="250.0"/>
    </flowset>
  </scenarios>
  <repository>
    <componentinstance id="ReadWheelSpeedSensors_Instance"
compatible="//@assembly/@component.0" variancePercentage="0.05">
      <service instance="//@assembly/@component.0/@service.0" cycles="600"
networkUsage="4000.0">
        <provide connects="//@assembly/@component.0/@outport.0"/>
        <depend>
          <require external="//@repository/@externalport.4"/>
          <require internal="//@assembly/@component.0/@inport.0"/>
        </depend>
      </service>
    </componentinstance>
    <componentinstance id="ControlWheelSpeed_Instance"
compatible="//@assembly/@component.1" variancePercentage="0.05">
      <service instance="//@assembly/@component.1/@service.0" cycles="500"
networkUsage="4000.0">
        <provide connects="//@assembly/@component.1/@outport.0"/>
        <depend>
          <require internal="//@assembly/@component.1/@inport.0"/>
        </depend>
      </service>
    </componentinstance>
    <componentinstance id="EngineVehicleInterface_Instance"
compatible="//@assembly/@component.2" variancePercentage="0.05">
      <service instance="//@assembly/@component.2/@service.0" cycles="500"
networkUsage="2000.0">
```

```
        <provide connects="//@assembly/@component.2/@outport.0"/>
        <depend>
          <require external="//@repository/@externalport.3"/>
          <require internal="//@assembly/@component.2/@inport.0"/>
        </depend>
      </service>
      <service instance="//@assembly/@component.2/@service.1" cycles="500"
networkUsage="2000.0">
        <provide connects="//@assembly/@component.2/@outport.1"/>
        <depend>
          <require internal="//@assembly/@component.2/@inport.1"/>
        </depend>
      </service>
      <service instance="//@assembly/@component.2/@service.2" cycles="500"
networkUsage="1000.0">
        <provide connects="//@assembly/@component.2/@outport.2"/>
        <depend>
          <require internal="//@assembly/@component.2/@inport.2"/>
        </depend>
      </service>
    </componentinstance>
    <componentinstance id="ProvidePowerModeInfo_Instance"
compatible="//@assembly/@component.3" variancePercentage="0.05">
      <service instance="//@assembly/@component.3/@service.0" cycles="400"
networkUsage="1000.0">
        <provide connects="//@assembly/@component.3/@outport.0"/>
        <depend>
          <require external="//@repository/@externalport.2"/>
        </depend>
      </service>
    </componentinstance>
    <componentinstance id="ControlEngineSpeedGauge_Instance"
compatible="//@assembly/@component.4" variancePercentage="0.05">
      <service instance="//@assembly/@component.4/@service.0" cycles="2850"
networkUsage="2000.0">
        <provide connects="//@assembly/@component.4/@outport.0"/>
        <depend>
          <require internal="//@assembly/@component.4/@inport.0"/>
        </depend>
      </service>
    </componentinstance>
    <componentinstance id="ControlVehicleSpeedGauge_Instance"
compatible="//@assembly/@component.5" variancePercentage="0.05">
      <service instance="//@assembly/@component.5/@service.0" cycles="2950"
networkUsage="2000.0">
        <provide connects="//@assembly/@component.5/@outport.0"/>
        <depend>
          <require internal="//@assembly/@component.5/@inport.0"/>
        </depend>
      </service>
    </componentinstance>
    <componentinstance id="Gauge_Engine_Instance"
compatible="//@assembly/@component.6" variancePercentage="0.05">
      <service instance="//@assembly/@component.6/@service.0" cycles="500">
        <depend>
          <require internal="//@assembly/@component.6/@inport.0"/>
        </depend>
      </service>
    </componentinstance>
    <componentinstance id="TransmissionVehicleInterface_Instance"
compatible="//@assembly/@component.7" variancePercentage="0.05">
      <service instance="//@assembly/@component.7/@service.0" cycles="100"
networkUsage="1000.0">
        <provide connects="//@assembly/@component.7/@outport.0"/>
        <depend>
          <require external="//@repository/@externalport.5"/>
        </depend>
      </service>
    </componentinstance>
    <componentinstance id="ControlGearSelectedIndication_Instance"
compatible="//@assembly/@component.8" variancePercentage="0.05">
      <service instance="//@assembly/@component.8/@service.0" cycles="2500"
networkUsage="1000.0">
        <provide connects="//@assembly/@component.8/@outport.0"/>
        <depend>
          <require internal="//@assembly/@component.8/@inport.0"/>
        </depend>
      </service>
    </componentinstance>
    <componentinstance id="Display_Engine_Instance" cost="55.0"
compatible="//@assembly/@component.9" variancePercentage="0.05">
      <service instance="//@assembly/@component.9/@service.0" cycles="500"
networkUsage="1000.0">
        <depend>
          <require internal="//@assembly/@component.9/@inport.0"/>
        </depend>
      </service>
      <service instance="//@assembly/@component.9/@service.1" cycles="500"
networkUsage="1000.0">
        <depend>
          <require internal="//@assembly/@component.9/@inport.1"/>
        </depend>
      </service>
      <service instance="//@assembly/@component.9/@service.2" cycles="500"
networkUsage="1000.0">
        <depend>
          <require internal="//@assembly/@component.9/@inport.2"/>
        </depend>
      </service>
      <service instance="//@assembly/@component.9/@service.3" cycles="500"
networkUsage="1000.0">
        <depend>
          <require internal="//@assembly/@component.9/@inport.3"/>
        </depend>
      </service>
    </componentinstance>
    <componentinstance id="ReadOATSensor_Instance"
compatible="//@assembly/@component.10" variancePercentage="0.05">
      <service instance="//@assembly/@component.10/@service.0" cycles="1000"
networkUsage="1000.0">
        <provide connects="//@assembly/@component.10/@outport.0"/>
        <depend>
          <require external="//@repository/@externalport.6"/>
        </depend>
      </service>
    </componentinstance>
    <componentinstance id="ControlOutsideAirTemp_Instance"
compatible="//@assembly/@component.11" variancePercentage="0.05">
```

```
      <service instance="//@assembly/@component.11/@service.0" cycles="2744"
networkUsage="1000.0">
        <provide connects="//@assembly/@component.11/@outport.0"/>
        <depend>
          <require internal="//@assembly/@component.11/@inport.0"/>
        </depend>
      </service>
    </componentinstance>
    <componentinstance id="ControlCoolantTempGauge_Instance"
compatible="//@assembly/@component.12" variancePercentage="0.05">
      <service instance="//@assembly/@component.12/@service.0" cycles="1500"
networkUsage="1000.0">
        <provide connects="//@assembly/@component.12/@outport.0"/>
        <depend>
          <require internal="//@assembly/@component.12/@inport.0"/>
        </depend>
      </service>
    </componentinstance>
    <componentinstance id="ReadDriverDoorAjarSwitch_Instance" cost="1.0"
compatible="//@assembly/@component.13" variancePercentage="0.05">
      <service instance="//@assembly/@component.13/@service.0" cycles="100"
networkUsage="1000.0">
        <provide connects="//@assembly/@component.13/@outport.0"/>
        <depend>
          <require external="//@repository/@externalport.0"/>
        </depend>
      </service>
    </componentinstance>
    <componentinstance id="ControlOdometer_Instance"
compatible="//@assembly/@component.14" variancePercentage="0.05">
      <service instance="//@assembly/@component.14/@service.0" cycles="2440"
networkUsage="4000.0">
        <provide connects="//@assembly/@component.14/@outport.0"/>
        <depend>
          <require internal="//@assembly/@component.14/@inport.0"/>
          <require external="//@repository/@externalport.7"/>
        </depend>
      </service>
    </componentinstance>
    <componentinstance id="ReadTripStemButton_Instance"
compatible="//@assembly/@component.15" variancePercentage="0.05">
      <service instance="//@assembly/@component.15/@service.0" cycles="100"
networkUsage="1000.0">
        <provide connects="//@assembly/@component.15/@outport.0"/>
        <depend>
          <require external="//@repository/@externalport.1"/>
        </depend>
      </service>
    </componentinstance>
    <componentinstance id="ReadLowWasherLevel_Instance"
compatible="//@assembly/@component.16" variancePercentage="0.05">
      <service instance="//@assembly/@component.16/@service.0" cycles="100"
networkUsage="1000.0">
        <provide connects="//@assembly/@component.16/@outport.0"/>
        <depend>
          <require external="//@repository/@externalport.8"/>
        </depend>
      </service>
    </componentinstance>
    <componentinstance id="ControlWasherLevelIndication_Instance"
compatible="//@assembly/@component.17" variancePercentage="0.05">
      <service instance="//@assembly/@component.17/@service.0" cycles="300"
networkUsage="1000.0">
        <provide connects="//@assembly/@component.17/@outport.0"/>
        <depend>
          <require internal="//@assembly/@component.17/@inport.0"/>
        </depend>
      </service>
    </componentinstance>
    <processor id="cpu066-h" clock="66.0" cost="100.0"
internalBusBandwidth="1024.0" internalBusDelay="0.1" lowerFail="0.015"
upperFail="0.03"/>
    <processor id="cpu066-l" clock="66.0" cost="140.0"
internalBusBandwidth="1024.0" internalBusDelay="0.1" lowerFail="0.01"
upperFail="0.025"/>
    <processor id="cpu100-h" clock="100.0" cost="125.0"
internalBusBandwidth="1024.0" internalBusDelay="0.1" lowerFail="0.015"
upperFail="0.03"/>
    <processor id="cpu100-l" clock="100.0" cost="175.0"
internalBusBandwidth="1024.0" internalBusDelay="0.1" lowerFail="0.01"
upperFail="0.025"/>
    <processor id="cpu133-h" clock="133.0" cost="150.0"
internalBusBandwidth="1024.0" internalBusDelay="0.1" lowerFail="0.015"
upperFail="0.03"/>
    <processor id="cpu133-l" clock="133.0" cost="210.0"
internalBusBandwidth="1024.0" internalBusDelay="0.1" lowerFail="0.01"
upperFail="0.025"/>
    <processor id="cpu166-h" clock="166.0" cost="175.0"
internalBusBandwidth="1024.0" internalBusDelay="0.1" lowerFail="0.015"
upperFail="0.03"/>
    <processor id="cpu166-l" clock="166.0" cost="245.0"
internalBusBandwidth="1024.0" internalBusDelay="0.1" lowerFail="0.01"
upperFail="0.025"/>
    <processor id="cpu200-h" clock="200.0" cost="200.0"
internalBusBandwidth="1024.0" internalBusDelay="0.1" lowerFail="0.015"
upperFail="0.03"/>
    <processor id="cpu200-l" clock="200.0" cost="280.0"
internalBusBandwidth="1024.0" internalBusDelay="0.1" lowerFail="0.01"
upperFail="0.025"/>
    <processor id="cpu233-h" clock="233.0" cost="225.0"
internalBusBandwidth="1024.0" internalBusDelay="0.1" lowerFail="0.015"
upperFail="0.03"/>
    <processor id="cpu233-l" clock="233.0" cost="315.0"
internalBusBandwidth="1024.0" internalBusDelay="0.1" lowerFail="0.01"
upperFail="0.025"/>
    <processor id="cpu266-h" clock="266.0" cost="250.0"
internalBusBandwidth="1024.0" internalBusDelay="0.1" lowerFail="0.02"
upperFail="0.035"/>
    <processor id="cpu266-l" clock="266.0" cost="350.0"
internalBusBandwidth="1024.0" internalBusDelay="0.1" lowerFail="0.015"
upperFail="0.03"/>
    <processor id="cpu300-h" clock="300.0" cost="275.0"
internalBusBandwidth="1024.0" internalBusDelay="0.1" lowerFail="0.02"
upperFail="0.035"/>
    <processor id="cpu300-l" clock="300.0" cost="385.0"
internalBusBandwidth="1024.0" internalBusDelay="0.1" lowerFail="0.015"
upperFail="0.03"/>
    <processor id="cpu333-h" clock="333.0" cost="300.0"
```

```xml
internalBusBandwidth="1024.0" internalBusDelay="0.1" lowerFail="0.02"
upperFail="0.035"/>
    <processor id="cpu333-l" clock="333.0" cost="420.0"
internalBusBandwidth="1024.0" internalBusDelay="0.1" lowerFail="0.015"
upperFail="0.03"/>
    <processor id="cpu366-h" clock="366.0" cost="325.0"
internalBusBandwidth="1024.0" internalBusDelay="0.1" lowerFail="0.02"
upperFail="0.035"/>
    <processor id="cpu366-l" clock="366.0" cost="455.0"
internalBusBandwidth="1024.0" internalBusDelay="0.1" lowerFail="0.015"
upperFail="0.03"/>
    <processor id="cpu400-h" clock="400.0" cost="350.0"
internalBusBandwidth="1024.0" internalBusDelay="0.1" lowerFail="0.02"
upperFail="0.035"/>
    <processor id="cpu400-l" clock="400.0" cost="490.0"
internalBusBandwidth="1024.0" internalBusDelay="0.1" lowerFail="0.015"
upperFail="0.03"/>
    <processor id="cpu433-h" clock="433.0" cost="375.0"
internalBusBandwidth="1024.0" internalBusDelay="0.1" lowerFail="0.02"
upperFail="0.035"/>
    <processor id="cpu433-l" clock="433.0" cost="525.0"
internalBusBandwidth="1024.0" internalBusDelay="0.1" lowerFail="0.015"
upperFail="0.03"/>
    <processor id="cpu466-h" clock="466.0" cost="400.0"
internalBusBandwidth="1024.0" internalBusDelay="0.1" lowerFail="0.02"
upperFail="0.035"/>
    <processor id="cpu466-l" clock="466.0" cost="560.0"
internalBusBandwidth="1024.0" internalBusDelay="0.1" lowerFail="0.015"
upperFail="0.03"/>
    <processor id="cpu500-h" clock="500.0" cost="450.0"
internalBusBandwidth="1024.0" internalBusDelay="0.1" lowerFail="0.02"
upperFail="0.035"/>
    <processor id="cpu500-l" clock="500.0" cost="630.0"
internalBusBandwidth="1024.0" internalBusDelay="0.1" lowerFail="0.015"
upperFail="0.03"/>
    <bus id="CAN-HS" bandwidth="500.0" delay="0.0020" cost="100.0"/>
    <bus id="CAN-MS" bandwidth="125.0" delay="0.0080" cost="50.0"/>
    <bus id="CAN-LS" bandwidth="33.3" delay="0.016" cost="25.0"/>
    <bus id="LIN" bandwidth="10.0" delay="0.05" cost="10.0"/>
    <externalport id="ajar-switch" lowerFail="0.01" upperFail="0.05"/>
    <externalport id="stem-button" lowerFail="0.01" upperFail="0.05"/>
    <externalport id="ignition-switch" lowerFail="0.01" upperFail="0.05"/>
    <externalport id="crankshaft-sensor" lowerFail="0.01" upperFail="0.05"/>
    <externalport id="wheel-sensor" lowerFail="0.01" upperFail="0.05"/>
    <externalport id="gear-sensor" lowerFail="0.01" upperFail="0.05"/>
    <externalport id="oat-sensor" lowerFail="0.01" upperFail="0.05"/>
    <externalport id="odometer-storage" lowerFail="0.01" upperFail="0.05"/>
    <externalport id="lowwasher-switch" lowerFail="0.01" upperFail="0.05"/>
  </repository>
  <objectives>
    <settings noRun="1" noSampling="50" maxCost="10000.0">
      <evaluations>Cost</evaluations>
      <evaluations>ResponseTime</evaluations>
      <evaluations>CPUUtilization</evaluations>
      <evaluations>Safety</evaluations>
    </settings>
  </objectives>
</aqosa.ir:AQOSAModel>
```