Cover Page

Universiteit Leiden

Leiden University Repository

The handle http://hdl.handle.net/1887/30105 holds various files of this Leiden University dissertation.

Author: Etemadi Idgahi (Etemaadi), Ramin
Title: Quality-driven multi-objective optimization of software architecture design : method, tool, and application
Issue Date: 2014-12-11

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

Ramin Etemaadi

November 2014

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

Proefschrift

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van Rector Magnificus prof.mr. C.J.J.M. Stolker,
volgens besluit van het College voor Promoties
te verdedigen op donderdag 11 december 2014
klokke 12.30 uur

door

Ramin Etemadi Idgahi
geboren te Teheran, Iran
in 1981

Promotion Committee

Promotor:	Prof. dr. Thomas H. W. Bäck	(Leiden University, the Netherlands)
Promotor:	Prof. dr. Michel R. V. Chaudron	(Chalmers and Gothenburg University, Sweden)
Co-promotor:	Dr. Michael T. M. Emmerich	(Leiden University, the Netherlands)
Other members:	Prof. dr. Joost N. Kok	(Leiden University, the Netherlands)
	Prof. dr. Paris Avgeriou	(University of Groningen, the Netherlands)
	Dr. Emilio Insfran	(University of Valencia, Spain)
	Dr. Todor Stefanov	(Leiden University, the Netherlands)

IPA Institute The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

OMECA Project Parts of this research were supported by the Dutch national project OMECA (Optimization of Modular Embedded Computer-vision Architectures).

Typeset by LaTeX.
ISBN: 978-94-6259-472-2

Copyright ©2014 Ramin Etemaadi

خیام اگر ز باده مستی خوش باش

با ماهرخی اگر نشستی خوش باش

چون عاقبت کار جهان نیستی است

انگار که نیستی چو هستی خوش باش

Khayyam, if you are intoxicated with wine, enjoy!

If you are seated with a lover of thine, enjoy!

In the end, the void the whole world employ

Imagine thou art not, while waiting in line, enjoy!

Contents

A man practices the art of adventure when he breaks the chain of routines.

Chapter 1

Introduction

In this chapter the motivations, the objectives and the contributions of this study are discussed.

This chapter is structured as follows. Section 1.1 gives a brief background on software architecture optimization and describes the contributions of this dissertation. Section 1.2 introduces the research objectives of this study and the research questions which we are going to address through this dissertation. After that, Section 1.3 discusses the outline of the dissertation for the rest of chapters.

1.1 Problem Statement and Contribution

Software architecting is a people-intensive, non-trivial and demanding task for software engineers to perform. At the same time, its architecting is a fundamental activity of software development because it involves several questions such as balancing the dependencies among components, maximization modularity, and fulfilling of quality requirements.

The architecture is a key enabler for software systems. Besides being crucial for user functionality, the software architecture [1] has deep impact on non-functional properties such as performance, safety, energy consumption and cost. Moreover, software architecture addresses fundamental design choices that are often difficult or very expensive to change in later stages of development. Hence, methods and techniques are needed

[1]It should be noted that we have approached the problem of architecture design coming from the area of software architectures. However, for most practical cases (including all case studies in this dissertation) an architecture comprises a hardware architecture and a software architecture. A more precise terminology would be to talk about 'system architecture' (as is common in system engineering).

In this dissertation, the term ***'software architecture'*** generally denotes the combination of hardware and software architecture.

for designing good software architectures which meet various quality constraints in the the early phases of development. The complexity of systems has increased sharply because of customers' demand for more user functions. Fierce competition makes short time-to-market important. To construct a system that optimizes all its requirements simultaneously is difficult, if not impossible. Therefore, architects are faced with a complex optimization problem. During optimization, a large design space needs to be searched in an efficient manner.

In the recent years, researchers have proposed automated approaches for supporting architects in creating architectural designs: (1) Rule-based approaches and (2) Meta-heuristic-based approaches [MKBR10]. Rule-based approaches try to detect weak points (e.g. bottlenecks) in the architectural model based on predefined rules and apply predefined solutions (tactics, patterns) to alleviate these weak points. Meta-heuristic approaches [BR03] frame the challenge of designing architectures as an optimization problem and iteratively try to improve a candidate solution with regard to a given measure of quality. In this way these algorithms explore very large design spaces to find optimal architectural solutions. The component-based paradigm makes it possible to easily and automatically create variation of architectural designs. Hence, the component-based paradigm is key to meta-heuristic optimization approaches for architecture design.

Evolutionary Algorithms (EA), as a recent meta-heuristic approach, are a common optimization technique for solving software architectural problems. However, EA for generating new solutions uses generic search operators, such as *Crossover* or *Mutate*, which are blind to the problem and do not take into account the domain knowledge. To overcome this issue, domain-specific search operators have been proposed by the researchers. The downside of using domain-specific search operators, is that the algorithm might find local optimal solutions. In addition, each domain-specific operator is usually useful only for one specific objective, and this is a threat to the optimality of results in multi-objective problems.

The Software Product Line (SPL) approach focuses on the development of specific products within a well-defined domain by leveraging their commonalities and managing their variabilities in a systematic way in order to obtain large-scale reuse [vdLSR07]. The SPL development paradigm is an approach for developing variant-rich software systems that has been widely adopted in recent years. A SPL is defined as a set of software systems with common managed features [CN01]. Software product line engineering aims to develop these systems by taking advantage of the massive reuse of core assets in order to improve time to market and product quality. In the SPL context, every product on the product lines has its own characteristics, and therefore it has different quality attributes. However, the architectural design is a critical factor SPLs because, ideally, the architecture forms a common basis on top of which different products can be build. The architecture has deep impact on non-functional properties such as performance, safety, reliability, security, energy consumption and cost.

In the context of software product lines, architects need to design an architecture that can work with different components that offer the same functionality, but differ in their behaviour, performance and other quality characteristics, and hence lead to different overall quality of the system.

In this dissertation, an automated approach for software architecture design is proposed that supports analysis and optimization of multiple quality attributes both for single products as well as for product lines.

The main contributions of this dissertation are:

- the demonstration of a meta-heuristic optimization approach for automated software architecture design in a real industrial system. More specifically, it reports the results of applying our architecture optimization framework to an automotive sub-system that was conducted based on a large-scale real world industrial case study. The framework supports multiple quality attributes including response time, processor utilization, bus utilization, safety and cost.

- the introduction of two novel degrees of freedom for the optimization of software architectures. It presents the usefulness of the topology degree of freedom as well as replication-degree of freedom. It demonstrates how the number of processing nodes and their interconnecting network can be codified to fit into a genetic algorithm genotype and thus be subject to automated synthesis. Our studies show that these extra degrees of freedom lead to better overall software architecture optimization. Moreover, it analyses the effectiveness of these two new degrees of freedom by running a very computationally-intensive experiment against our industrial case study. The results of this case study show us additional evidence for the usefulness of these two novel degrees of freedom.

- a comparison between various combinations of EA search operators (both domain-specific and generic) for multi-objective optimization of software architecture. The domain-specific operators we study are motivated by software architectural anti-patterns. However, each heuristic-based search operator improves only one quality attribute of the solution, which is challenging for multi-objective problems. To address this issue, we develop strategies for mixing generic and domain-specific search operators in evolutionary algorithms that speed up the finding of good solutions.

- the introduction of a new search-based approach for generating a set of optimal software architectural solutions for use in software product lines. This approach extends our architecture optimization framework in the direction of features (which can be considered to exist in the requirement-domain). In our new approach, we add feature models as input to the framework and take into account the relationship between the software components in the architecture and features in the feature model. Our proposed optimizer addresses this by first searching for architectures that are optimal for individual product. After that, it analyses the commonality of the found optimal solutions and proposes a set of solutions which are suitable for the range of products defined by various feature combinations.

1.2 Research Objectives

We formalize the objectives of the current work via the following research questions:

- **RQ1**

  Can meta-heuristic optimization improve the process of designing efficient architectures for a set of given quality attributes in an industrial domain?

- **RQ2**

  Can enlargement of the optimization search space help the meta-heuristic approach to find better architectural solutions?

- **RQ3**

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

- **RQ4**

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

1.3 Dissertation Outline

The rest of this dissertation is structured in these chapters: Chapter 2 introduces and defines common terminologies, Chapter 3 discusses related work, Chapter 4 introduces our proposed AQOSA framework, Chapter 5 presents three case studies, Chapter 6 introduces two novel degrees of freedom, Chapter 7 proposes problem-specific search operators, Chapter 8 introduces a new search-based approach for integration of feature models and architecture optimization together, Chapter 9 presents a parallel implementation of our framework, Chapter 10 concludes our findings. Brief summaries of these chapters are given next:

Chapter 2 defines the foundations on which this dissertation is built and introduces the used terminology. This chapter discusses (i) optimization and more specifically multi-objective optimization problems, (ii) software architecture, software component and component-based software architecture, (iii) quality of software architectures, and (iv) software product lines.

Chapter 3 discusses the state of the art of tools, methods, and approaches related to our proposed framework. The related work in this chapter is categorized into four categories: (i) the related optimization tools which tackle the software architecture optimization problem, (ii) industrial case studies which use an optimization approach in embedded systems, (iii) related heuristic-based approaches for software architecture optimization, (iv) search-based approaches and techniques in the domain of software product lines.

Chapter 4 introduces a new meta-heuristic optimization framework for automated software architecture design. The framework has been developed as a new implementation from scratch and is named ***AQOSA*** for ***A***utomated ***Q***uality-driven ***O***ptimization of ***S***oftware ***A***rchitectures. This chapter discusses the design of this framework.

Chapter 5 presents three software architecture design problems to which we apply our AQOSA framework. These cases show the effectiveness and usefulness of an automated quality-driven approach for the problem of software architecture design. These case studies will be the basis for the experiments in the following chapters, as well. To the best of our knowledge this is the largest case study on architecture optimization of real industrial embedded software system.

Chapter 6 introduces two novel degrees of freedom for the optimization of software architectures. We know that meta-heuristic approaches, such as genetic algorithms (GA), use degrees of freedom to automatically generate new alternative solutions. The two degrees of freedom are: (1) the topology of hardware platform, and (2) replication of software components. They can improve the results of the optimization algorithms by enlarging the design space. This chapter analyses the effectiveness of these two new degrees of freedom by running a very computationally-intensive experiment against an industrial case study.

Chapter 7 proposes an approach to make the optimization process faster by employing problem-specific search operators. The chapter discusses the use of architectural patterns and anti-patterns as heuristic-based search operators to reach the optimal solution faster. It also introduces different ways of combining the aforementioned search operators. In this chapter, an experiment was set up to compare various combinations of heuristic-based search operators for an embedded system architecture problem with multiple objectives based on a performance measurement called Averaged Hausdorff distance.

Chapter 8 proposes a new search-based approach for generating an optimal software architectural solution that supports a family of products that exist in the context of Software Product Line (SPL). This novel search-based method produces a set of solutions which are suitable for a range of products defined by various feature combinations. In this SPL-aware method, AQOSA will also consider feature models as input to the framework and take into account the relationship between the software components in the architecture and features in the feature model. Hence, the approach applies optimization techniques to each product of the SPL. After that, it analyses the commonality of the optimal solutions and proposes a set of solutions which are suitable for the entire range of products defined by various feature combinations.

Chapter 9 deals with making the optimization process faster by parallelising the execution. We know that meta-heuristic approaches in multi-objective problems especially for high dimensions mostly take a very long time to be executed. One of the best solutions to speed up this process is by parallelising execution of evolutionary algorithm on multiple computing nodes. This chapter presents the results of parallel execution of evolutionary algorithm for multi-objective optimization of software architecture. This chapter studies two ways for parallelising the execution of evolutionary algorithm.

Chapter 10 closes with a conclusion and suggestions for future work.

Appendix Last, but not least, in Appendix A a sample of AQOSA IR is presented. It shows the source of the AQOSA IR model of one of our case studies. It is encoded in the Eclipse EMF format.

Chapter 2

Definitions

This chapter defines the foundations on which this dissertation is built and introduces the basic concepts used terminology.

This chapter is structured as follows. Section 2.1 defines the optimization problem and more specifically multi-objective optimization problem. It also gives the definition of Pareto dominance for ordering of solutions in multi-objective optimization solutions and the Pareto front as the result of the optimization process. Section 2.2 defines software architecture, software component and component-based software architecture. These are very critical aspects of our approach and in our framework. The notion of quality of software architecture is described in Section 2.3 which is a fundamental concept in our optimization approach. Finally, Section 2.4 defines software product line, feature and product which are used in Chapter 8.

2.1 Evolutionary Multi-objective Optimization (EMO)

According to Bäck [Bäc96], Evolutionary Algorithms are based on a model of natural, biological evolution, which was formulated for the first time by **Charles Darwin**. The "Darwinian theory of evolution" explains the adaptive change of species by the principle of *natural selection*, which favours those species for survival and further evolution that are best adapted to their environmental conditions. In addition to selection, the other important factor for evolution is the occurrence of small, apparently random and undirected variations between the phenotypes. These mutations prevail through selection, if they prove their worth in light of the current environment; otherwise, they perish.

2.1.1 Genotype and Phenotype

According to [Bäc96] and [Mit98], the Evolutionary Algorithms are based on *genes* as transfer units of heredity. Genes are occasionally changed by mutations. The genotype of an individual using bit strings is simply the configuration of bits in that individual's chromosome. Selection acts on the individual (the individual is the unit of selection), which express in its phenotype the complex interaction within its *genotype*; i.e., its total genetic information, as well as the interaction of the genotype with the environment in determining the *phenotype*. The evolving unit is the *population* which consists of a common gene pool included in the genotypes of the individuals.

One of the distinctive features of the Genetic Algorithm (GA) approach is to allow the separation of the representation of the problem from the actual variables in which it was originally formulated. In line with biological usage of the terms, it has become customary to distinguish the 'genotype' - the encoded representation of the variables, from the 'phenotype' - the set of variables themselves.

2.1.2 Optimization Problem

According to Kruisselbrink [Kru12], an optimization problem is a triple (X, F, G), where:

- X is the search space, which is the non-empty set of all possible solutions.
- $F = \{f_1, \dots, f_k\}, k \in \mathbb{N}$, is a set of one or more objective functions that are to be minimized. Each objective function is a function of the form $f : X \to \mathbb{R}$ that maps elements of the search space to a score value.
- $G = g_1, \dots, g_p, p \in \mathbb{N}$, is a set of constraint functions that need to be satisfied. Each constraint function is of the form $g : X \to \mathbb{R}$ mapping elements of the search space to a constraint value. For a certain input $x \in X$ a constraint g is said to be satisfied if and only if $g(x) \geq 0$. Otherwise, if $g(x) < 0$, then solution x violates the constraint and is therefore infeasible.

2.1.3 Multi-objective Optimization Problem

A multi-objective optimization problem is an optimization problem with more than one objective function. For instance, for the triple (X, F, G), the set F consists of at least two objective functions [Kru12].

For multi-objective optimization problems, the definition of optimality is often based on the notion of Pareto dominance on the objective space. Pareto dominance introduces a partial order on the space of objective function values, being $\mathbb{R}^k$ for a problem with k objectives. In the context of minimization, this order is defined as the following section.

2.1.4 Pareto Dominance

We follow the definition of Kruisselbrink [Kru12] who defines Pareto dominance as follows. For any two vectors u and v:

u dominates v (notation $u \prec_{\mathrm{Pareto}} v$ or just $u \prec v$) iff:

$$\forall i \in \{1, \ldots, k\} : u_i \leq v_i \tag{2.1}$$

$$\text{and } \exists j \in \{1, \ldots, k\} : u_j < v_j \tag{2.2}$$

u weakly dominates v (notation $u \preceq v$) iff:

$$u \prec_{\mathrm{Pareto}} v \vee u = v \tag{2.3}$$

u strictly dominates v iff:

$$\forall i \in \{1, \ldots, k\} : u_i < v_i \tag{2.4}$$

u and v are incomparable (notation $u \parallel v$) iff:

$$u \npreceq v \wedge v \npreceq u \tag{2.5}$$

The partial order introduced by using the notion of Pareto dominance on the solution space can be used to define the goal of multi-objective optimization as to find Pareto optimizers:

2.1.5 Pareto Optimium

For a multi-objective optimization problem (X, F, G), a point x in X is called "efficient", if it is feasible with respect to G and there is no other point x' in X with $F(x') \prec_{\mathrm{Pareto}} F(x)$. The point $F(x)$ in the objective space is then called a Pareto optimum.

2.1.6 Pareto Front

For a multi-objective optimization problem (X, F, G), the set of all Pareto optima is called the "Pareto Front" and the set of all efficient points in X is called the "efficient set".

With the definition for a multi-objective optimization problem, and the goal to find either one, or multiple, Pareto optima, the basic notions of optimization have been introduced.

2.1.7 Hypervolume

A variety of quality indicators is used in order to measure the quality of Pareto front approximations. Among them, the hypervolume indicator is of outstanding importance. It is a quality indicator that rewards the convergence towards the Pareto front as well as the representative distribution of points along the front. The hypervolume measure was originally proposed by Zitzler and Thiele [ZT98], who called it the size of dominated space. Let Λ denote the Lebesgue measure, then the hypervolume measure (ϕ metric) is defined as:

$$\phi(B, y_{ref}) = \Lambda\left(\bigcup_{y \in B} \{y' \mid y \prec y' \prec y_{ref}\} \right), B \subseteq R^m \tag{2.6}$$

Here, $y_{ref} \in R^m$ denotes a reference point that should be dominated by all Pareto-optimal solutions.

2.2 Software Architecture and Software Components

2.2.1 Software Architecture

Numerous definitions for software architecture have been formulated, and the research community has not achieved final agreement on a common wording. In the following, we give two commonly accepted definitions for software architecture.

Software Architecture (Definition 1) A general definition, that is used in this dissertation, emphasizes design decisions:

> "A software system's architecture is the set of principal design decisions made about the system" [TMD10].

Interestingly, what is a principal design decision depends on the system goal. Examples names by Taylor et al. [TMD10] are the structure of the system, important decisions on functional behaviour, the interaction of components, and non-functional properties. This definition only mentions the core concept of design decision. It is independent of the question how these design decisions are formulated, and thus includes intangible software architectures that are not documented. Thus, this definition separates the software architecture from its representation.

Software Architecture (Definition 2) A commonly accepted definition of software architecture is given in ISO/IEC/IEEE standard 42010. A software system's architecture is:

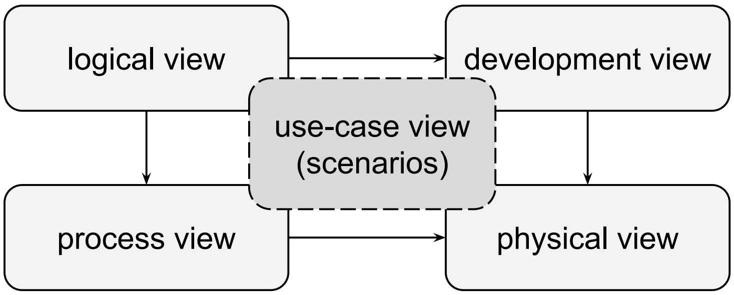


Figure 2.1: *The 4+1 view model of architecture*

> "[t]he fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution" [ISO11].

Because it concerns design at the software component level (as opposed to the design of those components themselves), software architecture is a pivotal vehicle to address and guarantee non-functional requirements. Since the interest in software architecture research has increased, several important concepts were introduced [Hei12].

For example, the influential 4 + 1-view model [Kru95] (as can be seen in Figure 2.1) expounded that, for representational clarity and the purpose of completeness, a software architecture is to be described according to predefined views. These views are defined so that they each accommodate the different issues that stakeholders have. The feedback that these stakeholders are then able to give is thought to benefit the fitness and other general design aspects of the software architecture.

2.2.2 Software Component

There term 'software component' is used with a somewhat different meaning in different fields of software engineering. In this section we first explain the view of components as result from decomposing the system. Next, we discuss software components as unit of composition.

An important subset of design decisions refer to the structure of the system, i.e., its decomposition into building blocks. To manage the complexity of software systems, architects apply the principles of encapsulation, abstraction and modularity to structure the system [TMD10]. The resulting building blocks are called software component: "A software component is an architectural entity that (1) encapsulates a subset of the system's functionality and/or data, (2) restricts access to that subset via an explicitly defined interface, and (3) has explicitly defined dependencies on its required execution context" [TMD10].

Researchers have strived to define a notion of software component which has as main objective to enable the composition of systems from independently developed components. Szyperski et al. [Szy98] have identified the following characteristics of a software component that can be independently developed and reused, stressing the composability and reuse by third parties:

Software Component A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties [Szy98].

According to Szyperski extended definition, a software component is:

- A subject for multiple use. A software component should be designed and implemented such that its functionality can be reused in many different systems.
- An externally stateless entity. A component should not expose its execution state to a system and can be bound, started and stopped at any moment of a system lifecycle.
- Composable with other components. A component provides well-specified interfaces, by which it can be bound to other components.
- An encapsulated entity, i.e. a component internal implementation cannot be explored through its interfaces.
- A unit of independent deployment. All component dependencies on external resources are clearly specified and it can be substituted by some other component.

Because component's aim to encapsulate their internal implementations from the outside world, they expose their functionality and connectivity specification via interfaces. A component interface is a set of named operations with specified signatures, through which it can interact with other components. As a special case, a component offers access to its functionality via its interfaces.

A component may have two types of interface: *provided* and *required* interfaces. Whereas a provided interface specifies the functionality that a component offers to the environment, a required interface specifies a component's requirements to the environment that have to be satisfied for its proper operation. More specifically, required interfaces are ports through which a component can invoke operations provided by other component interfaces. At component deployment, a required interface can be bound to a provided interface of another component.

Component Model A component model determines what is and what is not a component. Heineman and Councill [HC01] define a component model as follows:

> "A component model defines a set of standards for component implementation, naming, interoperability, customization, composition, evolution and deployment" [HC01].

This definition points out that a component model covers multiple facets of the development process, dealing with (i) rules for the construction of individual components, and (ii) rules for the assembly of these components into a system.

2.2.3 Component-Based Software Architecture

A software architecture that is structured based on software components and connectors is called a component-based software architecture in the following.

Component-Based Software Architecture A component-based software architecture is a software architecture whose principal design decisions regarding the structure of the systems are made by structuring the system as a set of software components. The system is thus described as a composition of components [Koz11].

To express (component-based) software architectures, architects have to describe the architecture in some type of artefact. These artefacts can be ranging from natural language descriptions over UML models [OMGa] to formal architectural description languages such as the Palladio Component Model [BKR09].

Architecture Model An architecture model is an artefact that captures some or all of the design decisions that compromise a system's architecture [TMD10].

Component-Based Architecture Model A component-based architecture model is a formal architecture model that uses software components as the main entity to describe the design decisions: (1) software component are explicit model entities which encapsulate internal decisions and provide information on interfaces and dependencies, (2) the model of a component can be reused in any CBA model, (3) structural design decisions are expressed as a composition and assembly of software components, only making use of the provided interfaces and context dependencies of the component models, and (4) other design decisions are described in relation to the composition or to the components (e.g. by annotating components, connectors, or assemblies) [Koz11].

2.3 Quality of Software Architecture

Developing high quality software products is a goal in many development projects. However, quality is a highly subjective term and depends on the goals and perceptions of stakeholders. The software architecture of a system is critical for a system to meet its quality-objectives. Thus, quality should be considered when designing software architecture.

To better reason about software product quality, software quality models have been suggested to describe and measure software quality, e.g. ISO/IEC 9126-1:2001 [ISO01]. (See [BKLW95] for more information)

Software quality attributes (also called quality characteristics or quality properties) are characteristics which provide the basis for evaluating quality of software systems. Examples of software quality attributes of software systems are performance, safety, reliability, maintainability, usability, and cost. Software quality attributes are one of the influencing factors to take into account when designing a software architecture [BCK03]. For some software quality attributes, quantitative quality metrics are available to assess the level of quality achieved by a software system.

Often, software quality objectives are found to be in conflict during the architectural design activity: For example, security and reliability often negatively influence each other: While a system is secure if it offers few places that keep sensitive data, such an organization may lead to single points of failure and decreased reliability. Furthermore, almost all software quality attributes conflict with performance [Koz11]. The art of architecting is to find suitable trade-off's between multiple mutually conflicting quality objectives.

2.4 Software Product Line (SPL)

The Software Product Line (SPL) approach is aimed at the development of a set of products within a well-defined domain by leveraging their commonalities and managing their variabilities in a systematic way in order to obtain large-scale reuse [vdLSR07].

2.4.1 Software Product Line

A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. Software product lines engineering (SPLE) is a software engineering paradigm institutionalizing reuse throughout the software life-cycle. Linden et al. [vdLSR07] define SPLE as a systematic approach for software reuse applied to a family of specific products within a well-defined domain.

For example, a TV set, as a consumer electronic device, uses software. In the modern TV production processes, SPL has been heavily employed. Various types of TV sets have lots of commonality in core functionalities but they might be vary in terms of internet connectivity, or capability of installing applications, or quality of display panel (e.g. HD, Full HD or 4K).

2.4.2 Feature

According to FODA(Feature-Oriented Domain Analysis) [KCH+90], "A feature is a prominent or distinctive and user-visible aspect, quality, or characteristic of a software system or systems". Feature models allow visualization, reasoning, and configuration of large and complex software product lines.

2.4.3 Product

Each product is formed by taking suitable features from the base of common assets, tailoring them as necessary through variation mechanisms, and then assembling the collection according to the rules of a common product line wide architecture [GH11]. Ideally, building a new product becomes more a matter of assembly or generation than one of creation: the predominant activity is integration rather than programming.

Chapter 3

State of the Art

In this chapter, the state of the art of tools, methods, and approaches are discussed which are related to our proposed framework and its related approaches in the following chapters. Besides the following related works from other research groups, the work presented in this dissertation is built on the previous works [BCdW06] and [LCL10].

In this chapter, related work is categorized into four categories which are structured as follows. Firstly, Section 3.1 details the related optimization tools which tackle the software architecture optimization problem. It should be noted that because each tool supports different objectives it is not practically feasible for us to directly compare the achieved results of our proposed framework with others. Then, in Section 3.2 some industrial case studies are listed which use an optimization approach in embedded systems. Although they use different optimization approaches at different levels, it is useful to compare our proposed approach with them. Section 3.3 discusses over related heuristic-based approaches for software architecture optimization. After that, a few approaches that use evolutionary algorithms to tackle software design problem are listed in Section 3.4. And lastly, Section 3.5 discusses different search-based approaches and techniques in the domain of software product lines.

3.1 Related Optimization Tools

3.1.1 PerOpteryx

Martens et al. [MKBR10] introduced an approach which can automatically improve software architectures based on trade-off analysis of performance, reliability, and cost by using a multi-objective genetic algorithm. The tool is a part of the Palladio project and is dedicated to the Palladio Component Model (PCM) [BKR09] for modelling and quality prediction. More technically, Layered Queuing Networks (LQN) methods are used for performance property prediction and Markov models are adopted for analysis of reliability. The Palladio component model (PCM) specifies component-based software architectures in a parametric way. Its simulation tool is capable of making performance predictions. Hence, this approach is suitable for systems which are modelled by PCM and focused on performance quality attributes. Instead, our approach is independent from any specific component model or modelling language.

3.1.2 ArcheOpteryx

ArcheOpteryx [ABGM09] [MBAG10] is a generic framework which optimizes architecture models by means of evolutionary algorithms. It works on the basis of AADL [FGH06] (Architecture Analysis & Design Language) specifications. Two quality criteria (data transmission reliability and communication overhead) are defined and the evaluation of architectures is based on analytical methods, rather than simulation. It supports only one degree of freedom for exploration, which is allocation (or deployment) of software components to hardware components. ArcheOpteryx specializes in probabilistic analysis of quality attributes and software architecture under uncertainty. Therefore, their tool and their analysis techniques are very suitable for situations with uncertainty. Their approach has different focus than ours, since they focus on probabilistic analysis and uncertainty conditions.

3.1.3 Multicube Explorer

Multicube Explorer [PSZ05] [YCAM$^+$11] proposes a design space exploration framework for supporting platform-based design. It allows optimization of parametrized system architectures. It presents a generic and structured framework for run-time resource management of embedded multi-core platforms. Based on the results of design-time multi-objective exploration, it defines a methodology to optimize the run-time allocation and scheduling of different application tasks. The design exploration flow results in a Pareto-optimal set of design alternatives in terms of power and performance trade-offs. The set of operating points can then be used at run-time to decide how the system resources should be distributed over different tasks running on the multiprocessor system on chip. So, it focuses on the link between design-time design space exploration and run-time configuration. However, our approach is not targeted for run-time optimization of configurations. The degrees of freedom for Multicube's optimization process mostly can be characterized as related to the configuration parameters (e.g. scheduling).

3.1.4 Artemis

Pimentel presented an overview of the Artemis workbench in [Pim08]. This workbench provides high-level modelling and simulation methods and tools for efficient performance evaluation and exploration of heterogeneous embedded multimedia systems. It consists of a set of methods and tools conceived and integrated in a framework to allow designers to model applications and System-on-Chip-based (multi-processor) architectures at a high level of abstraction, to map the former onto the latter, and to estimate performance through co-simulation of application and architecture models. It supports only one degree of freedom for exploration, which is allocation of software components.

3.1.5 KlaperSuite

The KlaperSuite [CFD+11] is a family of tools that offers designers a means to analyse performance and reliability at the architecture or design stage. The approach uses model transformations to automatically generate analysis models (queuing networks) out of an annotated UML design model. The approach focuses on validation of non-functional properties during early stages of design and does not offer optimization.

3.1.6 MOSES

MOSES [CPSV08] is a methodology proposed by Cortellessa et.al. which aims at the automated generation of feedback targeted at performance. It aims at systematically evaluating performance prediction results using step-wise refinement. It is built on top of UML-RT (UML for Real-Time) and presents a new implementation based on the UML2 meta-model. A unique feature of this approach is that it tries to diagnose the cause of performance problems in the system model level.

3.1.7 Other Related DSE Approaches

Hegedüs et al. [HHRV11] defined Design Space Exploration (DSE) as a process to analyse several "functionally equivalent" implementation alternatives, which meets all design constraints in order to identify the most suitable design choice (solution) based on quality metrics such as cost or dependability. They stated that in model-driven engineering, DSE is applied to find instance models that are (i) reachable from an initial model by a sequence of transformation rules and (ii) satisy a set of structural and numerical constraints. However, our approach using Genetic Algorithm (GA) approach does not start from any specific initial state. The initial population in GA can be spread out in the design space by means of randomized generation. Another difference is that, those approaches use a sequence of transformations to evolve the solution. But in GA, it is possible that a revolutionary solution is proposed as a result of some random mutation(s) in the genotype.

Räihä et al. in [RKM11] introduced a multi-objective genetic synthesis approach for software architecture. However, they cover two different quality attributes of software architectures; modifiability and efficiency. Therefore, although they use GA like our approach, we target different quality attributes.

3.2 Related Industrial Case Studies

3.2.1 Reliability Optimization for Embedded Systems

Glaß et al. [GLHT10] presented an automatic reliability-aware system-level design methodology to tolerate hardware defects. Real-life case studies from the automotive domain have been used to validate the effectiveness of the proposed techniques. For the reliability analysis and optimization at design time, a real-life specification for an adaptive light control has been explored. For the performance of the proposed online algorithm, a specification that combines six automotive applications including an adaptive cruise control and brake-by-wire system has been used. Similar to our case study, it studies real-life cases from the automotive domain. However, their optimization process tries to optimize two objectives: reliability and cost. Instead, our case study explores five objectives. Moreover, their approach does not use genetic algorithms.

3.2.2 Symbolic Multi-objective Design Space Exploration

Lukasiewycz et al. [LGHT08] formalized the problem of design space exploration as multi-objective 0—1 integer linear programming problem. They proposed a heuristic approach based on Pseudo Boolean (PB) solvers and a complete multi-objective PB solver based on a backtracking algorithm that incorporates the non-dominance relation from multi-objective optimization, called Heuristic multi-objective PB Solver (HPB). To emphasize the efficiency of the proposed algorithms, they applied the algorithms to a real adaptive light control from the automotive area. So, similar to the previous work, it comes from the automotive domain. However, their focus is on the proposed HPB approach and comparison with other algorithms, instead of the industrial case study. In contrast, our case study does not compare design space exploration algorithms.

3.2.3 Multiprocessor Systems-on-Chip Synthesis

In [CFL$^+$10] Ceriani et al. compared three evolutionary approaches for real-time, embedded, heterogeneous, multiprocessor systems (Multiprocessor Systems-on-Chip or

MP-SoCs). They considered three approaches: a multi-objective genetic algorithm, multi-objective Simulated Annealing, and multi-objective Tabu Search. In an experimental evaluation, they applied their method to a realistic JPEG compressor on the Xilinx FPGA toolchain. Again, similar to the previous work, although they worked on a real-life case study, their focus is on comparing three exploration algorithms.

3.2.4 Exploring Embedded System Architectures

Pimentel et al. [PEP06] presented the Sesame framework, which provides high-level modelling and simulation methods and tools for system-level performance evaluation and exploration of heterogeneous embedded systems. They demonstrated the framework by using a case study which traverses Sesame's exploration trajectory for a Motion-JPEG encoder application. They focused on multiple abstraction levels in their method. So, their framework allows architectural exploration at different levels of abstraction. A Motion-JPEG (M-JPEG) encoder application has been used as industrial case study to illustrate Sesame's modelling methodology and trajectory. Similar to our approach, they used a genetic algorithm for design space exploration. However, they focus on performance simulation. Instead, our approach aims to address other quality attributes in addition to performance as well.

3.3 Related Heuristic-based Approaches

3.3.1 Architectural Tactics

Koziolek et al. [KKR11] introduced a hybrid approach that incorporates architectural performance "tactics" into an evolutionary optimization process. They defined architectural tactics to improve two quality attributes: Performance and Cost. They implemented those tactics as part of the evolutionary optimization process. They showed that by using tactics, optimization algorithms can achieve better solutions. However, their experiment was conducted in an information systems context and considered 2-objective optimization.

Martens at el. in [MAK$^+$10] proposed another way of hybridization for optimization of software architectures. They propose two-step optimization: first analytic optimization and then a subsequent evolutionary optimization based on the Pareto candidates from the first step. However, this way is totally different from our approach. Because in our approach, we use the domain knowledge as the search operators and therefore, we integrated that knowledge as part of the evolutionary optimization process.

3.3.2 Anti-patterns in Palladio

Turbiani et al. [TK11] discussed the advantages of using software performance anti-patterns in an iterative manner. They introduced a couple of performance anti-patterns and defined an automatic approach to detect and solve the bottlenecks in software architectures. They demonstrated that, by applying this technique iteratively, the system performance can be improved significantly. However, they did not discuss quality attributes other than performance. They also did not integrate their approach within an evolutionary optimization process. Thus, the downside of their approach is that without having generic degrees of freedom and involving randomness in the optimization iterations, the optimality of the results is highly dependent on the initial architecture.

3.4 Software Design using Evolutionary Algorithms

3.4.1 Evolutionary Search in Object-Oriented Class Design

Simons et al. [SPG10] reported the findings of two experimental early life cycle design episodes wherein a human designer and software agents interact and collaborate to jointly steer an evolutionary, multi-objective search toward useful and interesting class designs. Their interactive search-based design approach takes place during upstream design before the design is realized in source code. It involves both a representation of the design problem and a representation of the design solution which are inherently traceable; and the representation of the solution space is a UML class model. Their approach changes the software design through an evolutionary process. Objective fitness functions available to the designer relate to the structural integrity of the solution class designs and include cohesion of classes, coupling between classes, and number of classes in a design. In their approach, they focus on interaction between human designer and software agents during evolutionary process which is different from our goals.

3.4.2 Class Responsibility with Genetic Algorithms

Bowman et al. in [BBL10] provided a decision-making approach to re-assign methods and attributes to classes in a class diagram. Their solution is based on a multi-objective genetic algorithm and uses class coupling and cohesion measurement for defining fitness functions. Their approach takes as input a class diagram to be optimized and suggests possible improvements to it. Their solution to the problem of class responsibility assignment in the context of object-oriented analysis and domain class models, suggests that the multi-objective genetic algorithm can help correct suboptimal class responsibility assignment decisions and perform far better than simpler alternative heuristics such as hill climbing and a single objective GA. However, our approach does not aim to optimize at the class diagram level. So, we have different focus from this study.

3.5 Related Software Product Lines (SPL)

3.5.1 Variability-driven Quality Evaluation in SPL

Etxeberria et al. in [EM08] proposed a novel method for evaluation of a product line. It is clear that the quality evaluation in software product lines is much more complicated than in single-systems because different products within the same product line can require different quality levels and the product line can have variabilities in its design that affect the quality. However, we know that the evaluation of all the products of a product family is very expensive. They presented a method for facilitating cost-effective quality evaluation of a product line taking into consideration variability on quality attributes.

They mentioned that the assessment of all the instances of the product line may not be worthwhile due to the high cost. However, it is possible to shorten product architecture evaluations because *"The product architecture evaluation is a variation of the product-line architecture evaluation as the product architecture is a variation of the product-line architecture and the extent to which product evaluation is a separate, dedicated evaluation depends on the extent to which the product architecture differs in quality-attribute-affecting ways from the product line architecture"* . Therefore, their approach is to create a generic evaluation model with variability that helps to evaluate the product family, thus reducing efforts, as several products can be evaluated together with that model.

3.5.2 Search-based Design of SPL

Colanzi in [Col12] proposed an approach for search-based design of the software product line architectures (PLA). She introduced a multi-objective optimization approach to the PLA design to ease the SPL development and to automate the PLA design. So, this approach focuses on search-based design of product lines; however in our approach, we assume that a product line is already designed and fixed. Then, we search to find optimal architectural solutions for the products of that fixed product line. Hence, our approach is not aimed at optimizing the features in a product line.

In another extended work, Colanzi et al. in [CV12] discussed the lessons they have leaned from applying search-based optimization to software product line architectures. They concluded that the results point out it is necessary to use SPL-specific measures and evolutionary operators more sensitive to the SPL context.

3.5.3 SPL Configuration

Sayyad et al. in [SIMA13] proposed an evolutionary algorithm, based on the Indicator-Based Evolutionary Algorithm (IBEA), for the problem of software product lines configuration. Specially, they targeted for large scale product lines. They mentioned that software product lines are hard to configure and techniques which work for medium sized product lines fail for much larger product lines such as the Linux kernel with 6000+ features. They found 30 sound solutions for this very large product line; however, we assume any arbitrary sound configuration as an input to our tool and our approach is not aimed at searching for sound configurations in a product line. In short, they showed an approach to automate the configuration of the very large variability models available from LVAT (Linux Variability Analysis Tools) feature model repository.

Moreover, Sayyad et al. in [SMA13] extended the aforementioned work to bring the user preferences in to the perspective. In this work, they presented a high-dimensional multi-objective search approach, which puts each user preference in focus without aggregation, and then incorporates the user preferences in the Pareto dominance criteria, using the Indicator-Based Evolutionary Algorithm (IBEA). They used 5 optimization objectives, namely: to maximize logical (syntactic) correctness, maximize richness of feature offering, minimize cost, maximize code reuse and minimize known defects. As can be seen, they are completely different from our approach objectives.

3.5.4 Software Evolution in SPL

Abrahão et al. in [AGHIR12] presented a framework to support the development and evolution of high-quality software product lines. The framework is based on several interrelated models or system views (eg, functionality, variability, quality) and a production plan defined by model transformations that generate a software system that meets both functional and quality requirements. The variability management involves the manipulation of features, represented as cardinality-based feature models, and the support of such variability in the so-called product line core assets. Specifications of the system variability, functionality, and quality can be dealt with models that are independent from each other but where their inter-consistency is assured by means of the relationships defined in this multi-model. The SPL production plan is parametrized by means of the multi-model which specifies the corresponding model transformations that are needed to obtain a specific product with the desired features, functions, and quality. Therefore, they focus on software product lines evolution by using a multi-model approach, which is completely different from our targets.

3.5.5 Quality Engineering for SPL

Kolb et al. in [KM06] discussed the quality engineering for software product lines. They provided an understanding on how to perform quality engineering and to systematically assure the quality of product lines and reusable artefacts. They discussed about how quality engineering for software product lines and generic software components needs to be different from traditional software systems and how quality engineering can be performed in the context of product lines. In addition to that, they provided a discussion of the difficulties and challenges of quality engineering for product lines and investigates the implications of product lines and reusable components on quality engineering. To summarize, their goal was to provide an understanding of existing constructive and analytical quality engineering techniques and methods and how these techniques and methods can be applied in a software product line context. The methods and techniques addressed include systematic architecture design and evaluation, architecture conformance checking, inspections, testing, metrics and measurement evaluation, and risk management. For each technique or method, it is discussed what the problems and challenges in the context of software product lines are, which benefits and limitations the technique or method has, and how it has to be extended to be applicable for assuring the quality of software product lines and reusable artefacts. Their approach can be considered as a the translation of classical manual software quality assurance techniques to the domain of software product lines.

Chapter 4

AQOSA Framework

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

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

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

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

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

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

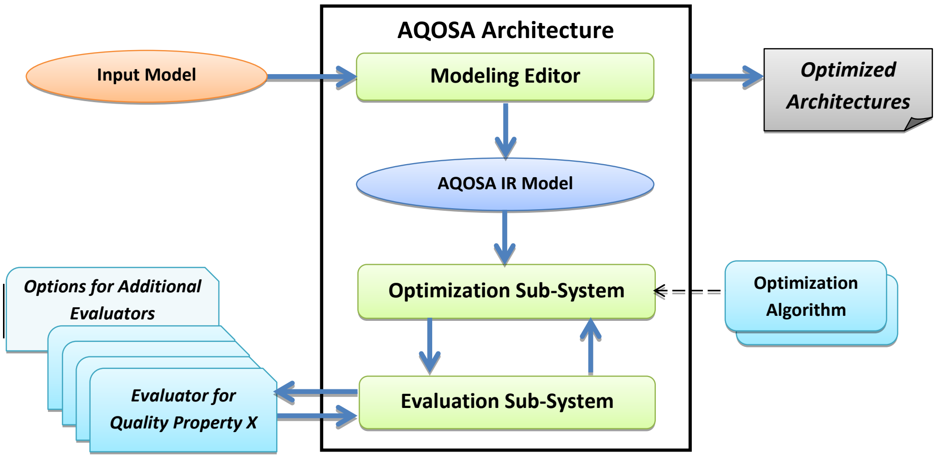


Figure 4.1: *AQOSA high level architecture*

4.1 Framework Overall Process

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

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

Then, AQOSA iterates through the following steps:

1. Generate a new set of candidate architecture solutions: To this end, AQOSA uses a representation of the architecture where it knows which are the degrees of freedom in the design and how to generate alternative architecture candidates.

2. Evaluate the new set of candidate architecture solutions for multiple quality properties: This works by generating analysis models from the architecture model using model transformations and then invokes up the property-specific analysers to evaluate these models.

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

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

4.2 AQOSA Tooling Design

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

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

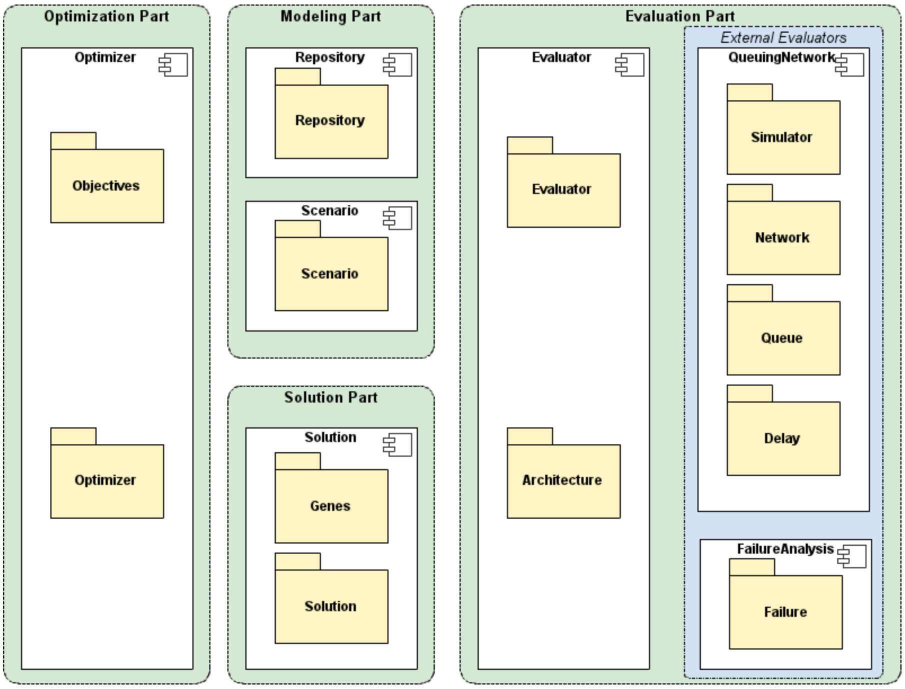


Figure 4.2: *AQOSA tooling parts*

2. `Optimization` part: Using evolutionary algorithms, this part tries to explore the design space for finding optimal solutions. It employs the Solution module to vary architectural solutions and the Evaluator module to measure them. More details are given in Section 4.4.

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

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

4.3 Modelling Part

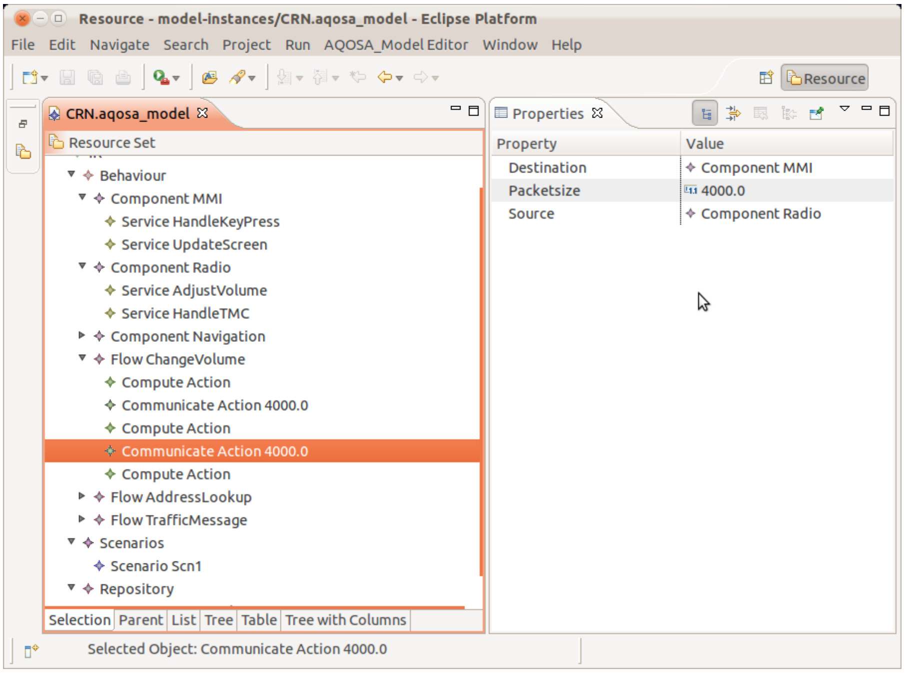
Because AQOSA is designed to optimize architectures in a wide range of domains, it aims to be independent from specific modelling languages. Hence, it uses its own internal architecture representation, AQOSA intermediate representation (AQOSA IR). AQOSA architecture modelling is defined by means of the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

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

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

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

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


Resource - model-instances/CRN.aqosa_model - Eclipse Platform
File Edit Navigate Search Project Run AQOSA_Model Editor Window Help
Resource
CRN.aqosa_model
Properties
Selection Parent List Tree Table Tree with Columns

```
Resource Set
  ▼ Behaviour
    ▼ Component MMI
        Service HandleKeyPress
        Service UpdateScreen
    ▼ Component Radio
        Service AdjustVolume
        Service HandleTMC
    ▶ Component Navigation
    ▼ Flow ChangeVolume
        Compute Action
        Communicate Action 4000.0
        Compute Action
        Communicate Action 4000.0
        Compute Action
    ▶ Flow AddressLookup
    ▶ Flow TrafficMessage
  ▼ Scenarios
      Scenario Scn1
  ▼ Repository
```

Property	Value
Destination	Component MMI
Packetsize	4000.0
Source	Component Radio

Selected Object: Communicate Action 4000.0

Figure 4.3: *AQOSA modelling tool screenshot*

- **Objectives**: This part defines the objectives which the framework optimizes.

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

4.4 Optimization Part

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

The problem of optimizing software architecture is a non-linear and discontinuous problem, i.e. small changes in the architecture design can have a very large impact

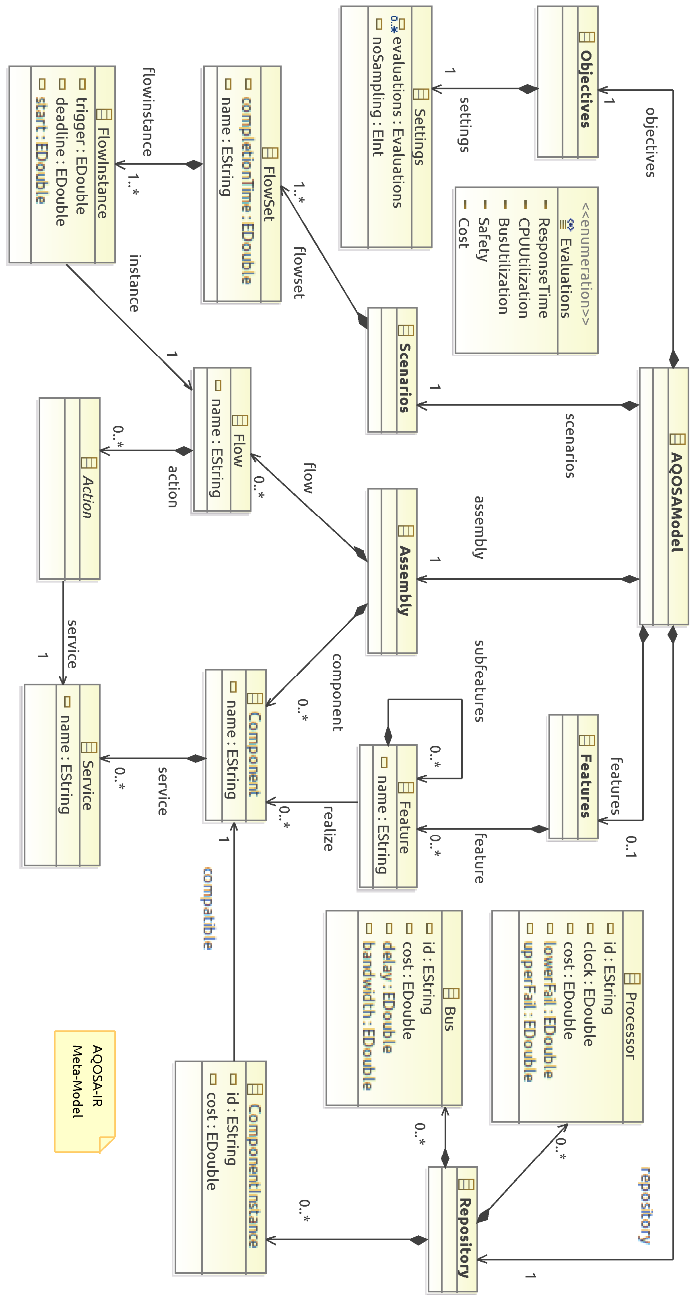


Figure 4.4: *AQOSA Intermediate Representation (IR) simplified meta-model*

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

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

4.4.1 Evolutionary Algorithms

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

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

- SPEA2 (an improved version of Strength Pareto Evolutionary Algorithm): It has been suggested by Zitzler and Thiele [ZLT02], and it is also widely used. It uses alternative ways for convergence and diversity compared to NSGA-II.

- SMS-EMOA (S-Metric Selection Evolutionary Multi-Objective Algorithm): It has been proposed by Emmerich, Beume and Naujoks [EBN05] [BNE07]. It is a representative of the class of hypervolume-based EMOA, which recently gained popularity in the EMOA field.

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

4.4.2 Degrees of Freedom

When an architectural design is created, usually, there are still variation to the solution without changing the functionality. We call them *Degrees of Freedom (DoF)*. The component-based paradigm that underlies our approach, allows us to recompose components in different topologies and wrappers. However, the optimizer should consider only the variations of architectural designs which do not modify the interfaces used in the architecture in order to guarantee that the optimization process does not change the functionality of the system.

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

Number of hardware nodes

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

Number of connections between hardware nodes

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

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

Network topology

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

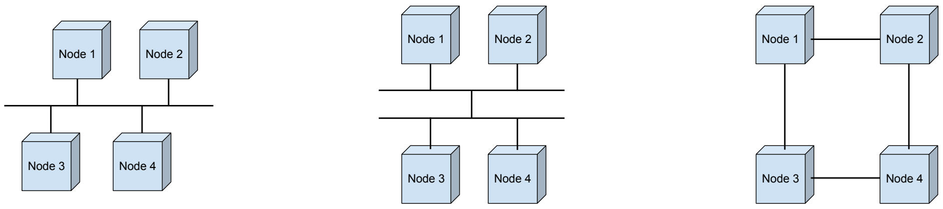


Figure 4.5: *Three possible topologies for a 4-node network*

Software on hardware allocation

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

Software components selection

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

Hardware components selection

This DoF entails that each hardware node can be replaced by another hardware node in the repository. These nodes may be different in processing speed, energy consumption, failure probability and cost. Hence, it has large effect on all quality aspects.

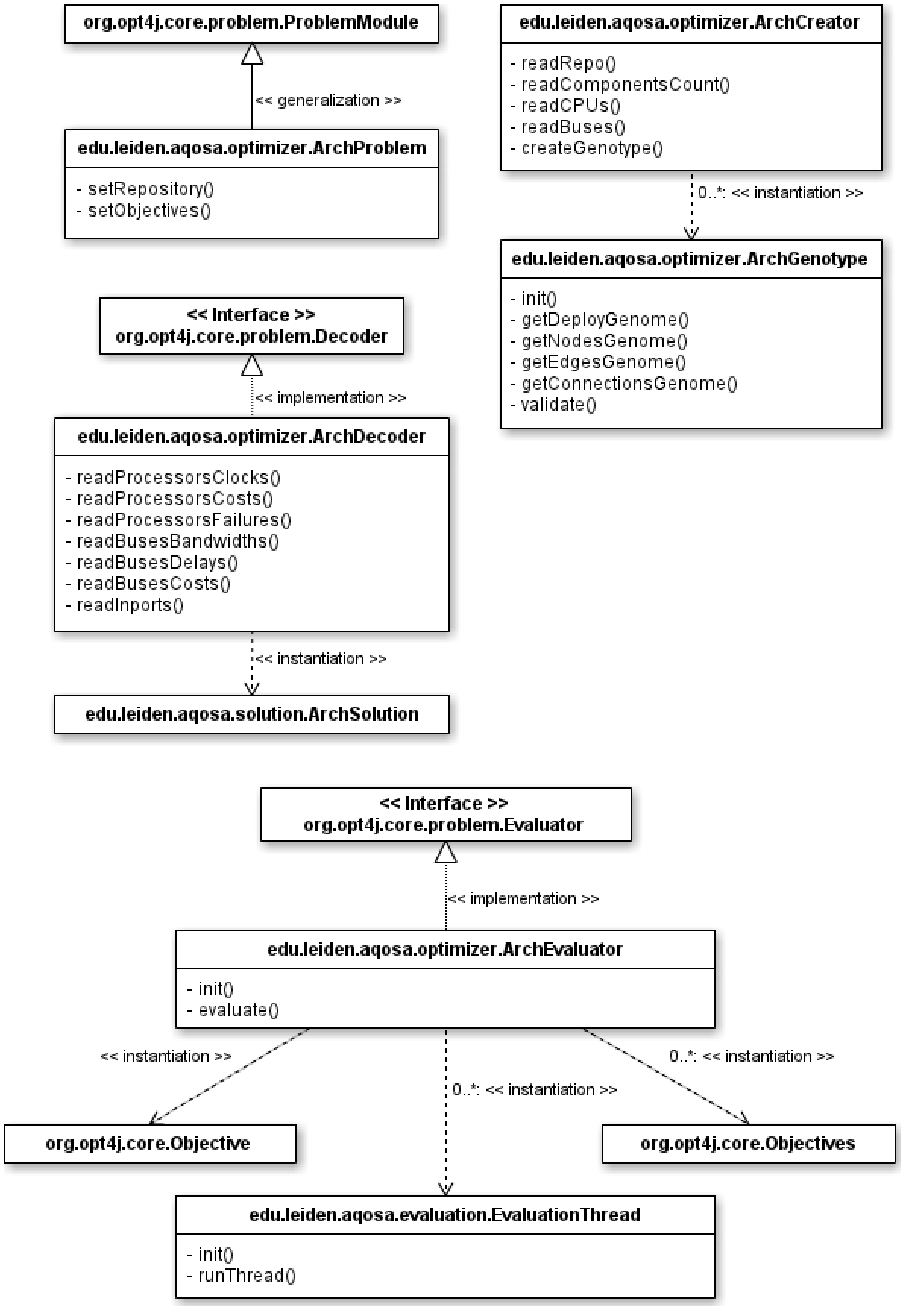


Figure 4.6: *Optimizer module class diagram*

Communication lines selection

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

4.4.3 Optimizer Module Implementation

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

4.5 Solution Part

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

4.5.1 Architecture Genotype Structure

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

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

2. The *nodes genome* (Table 4.1b) represents which hardware variant is selected for each node in the system. It encodes 'hardware components selection' DoF. The specification for each hardware node includes processing clock rate, range of failure probability, and cost.

3. The *communication genome* (Table 4.1c), like the nodes genome, represents which hardware variant is selected for each communication line. It encodes the 'number of connections' and the 'communication lines selection' DoFs. Their specification includes bandwidth, transmit delay, and cost.

Table 4.1: *AQOSA framework genotype*

Component$_1$	**Component$_2$**	**Component$_3$**	...	**Component$_c$**
Implmnt. $< i_1 >$ deployed on Node n_{i_1}	Implmnt. $< i_2 >$ deployed on Node n_{i_2}	Implmnt. $< i_3 >$ deployed on Node n_{i_3}	...	Implmnt. $< i_c >$ deployed on Node n_{i_c}

(a) *Deploy Genome*

n	**Node$_1$**	**Node$_2$**	**Node$_3$**	...	**Node$_n$**
(No. of Nodes)	HW Spec. $< h_1 >$	HW Spec. $< h_2 >$	HW Spec. $< h_3 >$	...	HW Spec. $< h_n >$

(b) *Nodes Genome*

l	**Line$_1$**	**Line$_2$**	**Line$_3$**	...	**Line$_l$**
(No. of Lines)	Bus Spec. $< b_1 >$	Bus Spec. $< b_2 >$	Bus Spec. $< b_3 >$	...	Bus Spec. $< b_l >$

(c) *Communication Lines Genome*

	Node$_1$	**Node$_2$**	**Node$_3$**	...	**Node$_n$**
Line$_1$	True/False	True/False	True/False	...	True/False
Line$_2$	True/False				True/False
Line$_3$	True/False		**Buses-to-Nodes Connection Matrix**		True/False
...	True/False				True/False
Line$_l$	True/False	True/False	True/False	...	True/False

	Line$_1$	**Line$_2$**	**Line$_3$**	...	**Line$_l$**
Line$_1$	True/False	True/False	True/False	...	True/False
Line$_2$	True/False				True/False
Line$_3$	True/False		**Buses-to-Buses Connection Matrix**		True/False
...	True/False				True/False
Line$_l$	True/False	True/False	True/False	...	True/False

(d) *Connections Genome*

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

4.5.2 Genotype Validation

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

4.5.3 Solution Module Implementation

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

4.6 Evaluation Part

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

The AQOSA framework uses analysis tools for particular quality attributes as plug-ins. It is assumed that these analysis tools are developed and validated by domain experts. However, to examine the accuracy of AQOSA evaluation implementation, the results generated by AQOSA evaluation have been compared with results reported in relevant publications. In case of performance attributes, the QN implementation

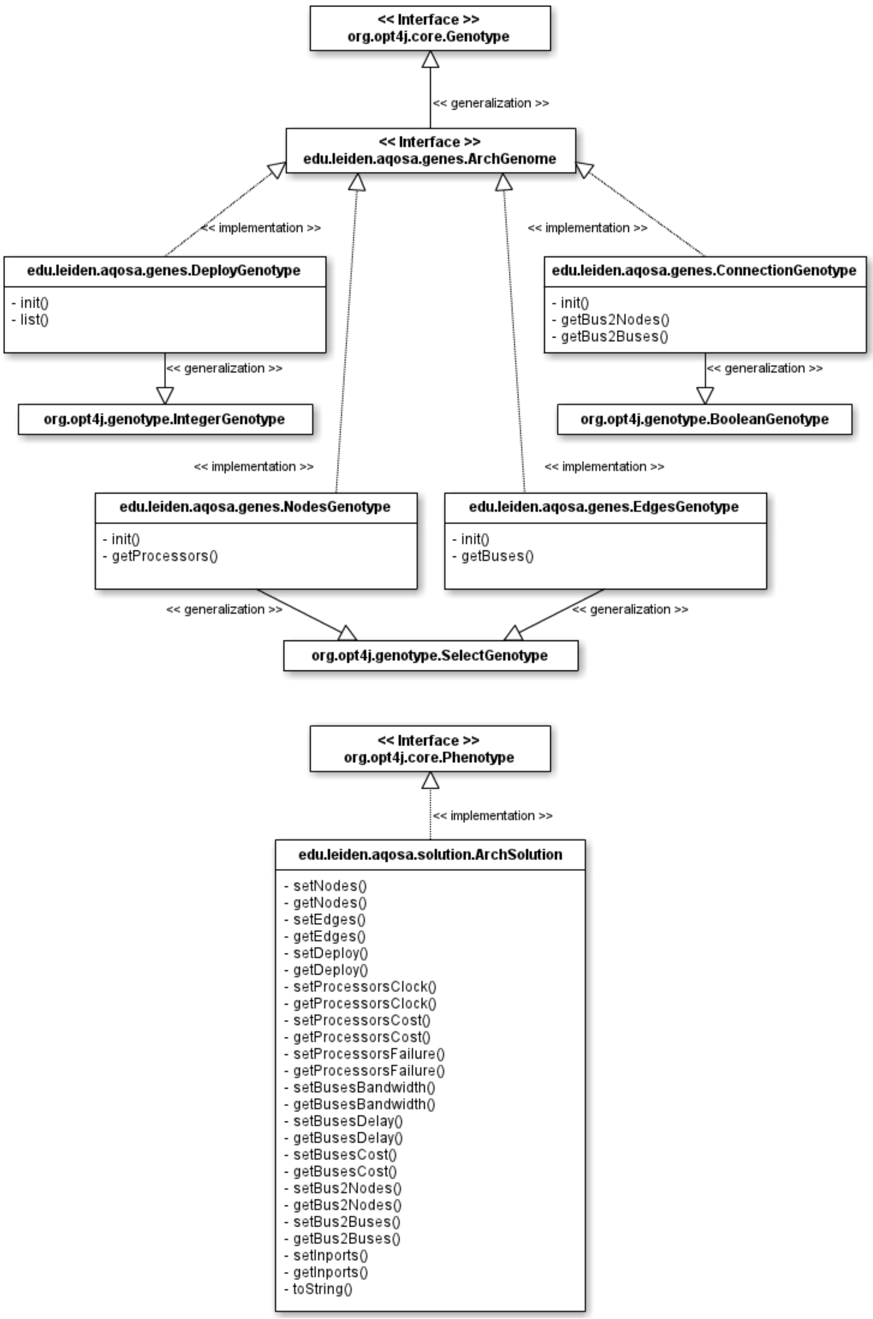


Figure 4.7: *Solution module class diagram*

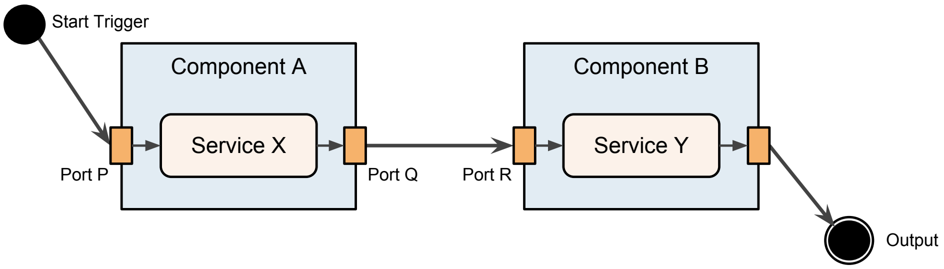


Figure 4.8: *Example scenario*

can achieve the same results as the results published in [WTVL06]. In case of FTA, the results have been compared with the results published in [FS10].

4.6.1 Evaluation Model Transformation

To explain how the transformation of an architectural solution to an evaluation model works, in the following two examples are given: (1) for queuing network transformation, (2) for fault tree transformation. For the purpose of the example, a very simple scenario is assumed which is depicted in Figure 4.8. The figure shows a software architecture that consists of two components: *Component A* and *Component B*. The start trigger calls *Service X* from *Component A* through *Port P*. Then, *Service X* passes data through *Port Q* to *Service Y* from *Component B* which receives the data through *Port R*. Subsequently, *Service Y* generates the required data as the output of the scenario.

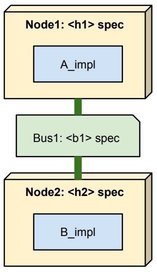
Figure 4.9: *Sample genotype*

ComponentA	**ComponentB**
A_{impl} deployed on $Node_1$	B_{impl} deployed on $Node_2$

2 (No. of Nodes)	**Node₁**	**Node₂**
	HW Spec. $< h_1 >$	HW Spec. $< h_2 >$

1 (No. of Lines)	**Line₁**
	Bus Spec. $< b_1 >$

	Node₁	**Node₂**
Line₁	True	True



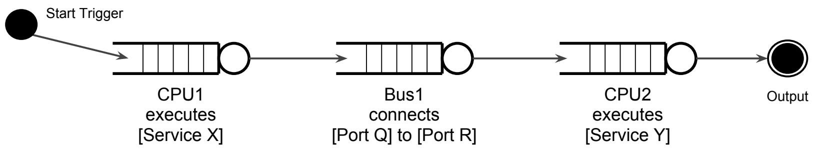

Start Trigger
CPU1 executes [Service X]
Bus1 connects [Port Q] to [Port R]
CPU2 executes [Service Y]
Output

Figure 4.10: *Generated queuing network for sample scenario*

In addition to this scenario, for evaluating an architecture based on a queuing network it is needed to know the hardware topology and the deployment of the candidate solution as well. Assume, the solution is a simple architecture, which contains two processing nodes and one bus which connects them together. Table 4.9 represent that architecture based on the AQOSA genotype structure. *Component A* has been deployed on Node_1 and *Component B* on Node_2. In this case, Bus_1 represents the connection between *Port Q* and *Port R*.

Queuing Network Transformation

To generate a Queuing Network (QN), the AQOSA evaluation part will create a QN as shown in Figure 4.10. Each processing node or communication line represents one resource. Hence, each is mapped to its own queue. In this network, *CPU1* can be taken by *Service X* (because *Component A* has been deployed on Node_1), *CPU2* can be taken by *Service Y* and *Bus1* can be busy when *Service X* passes the data to *Service Y*.

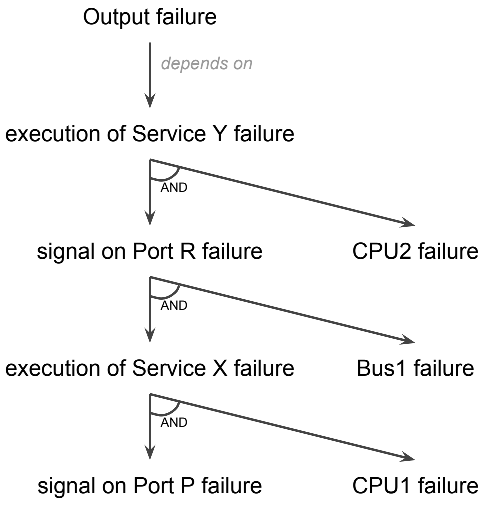


Figure 4.11: *Generated fault tree for sample scenario*

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

Fault Tree Transformation

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

4.6.2 Quality Attributes

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

Response Time

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

Processor Utilization

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

Communication Line Utilization

Like processor utilization, communication line utilization is the percentage of bandwidth that a communication line (or a bus) uses. Similar to above, AQOSA measures this metric for each bus in the architecture individually. Again, this metric can be configured to return either average, minimum, or maximum bus utilization.

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

System Safety

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

System Cost

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

Extension of Quality Attributes

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

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

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

For example, it is easily possible to extend AQOSA for **power consumption quality attribute**. To this end, it is needed to implement a transformation component which:

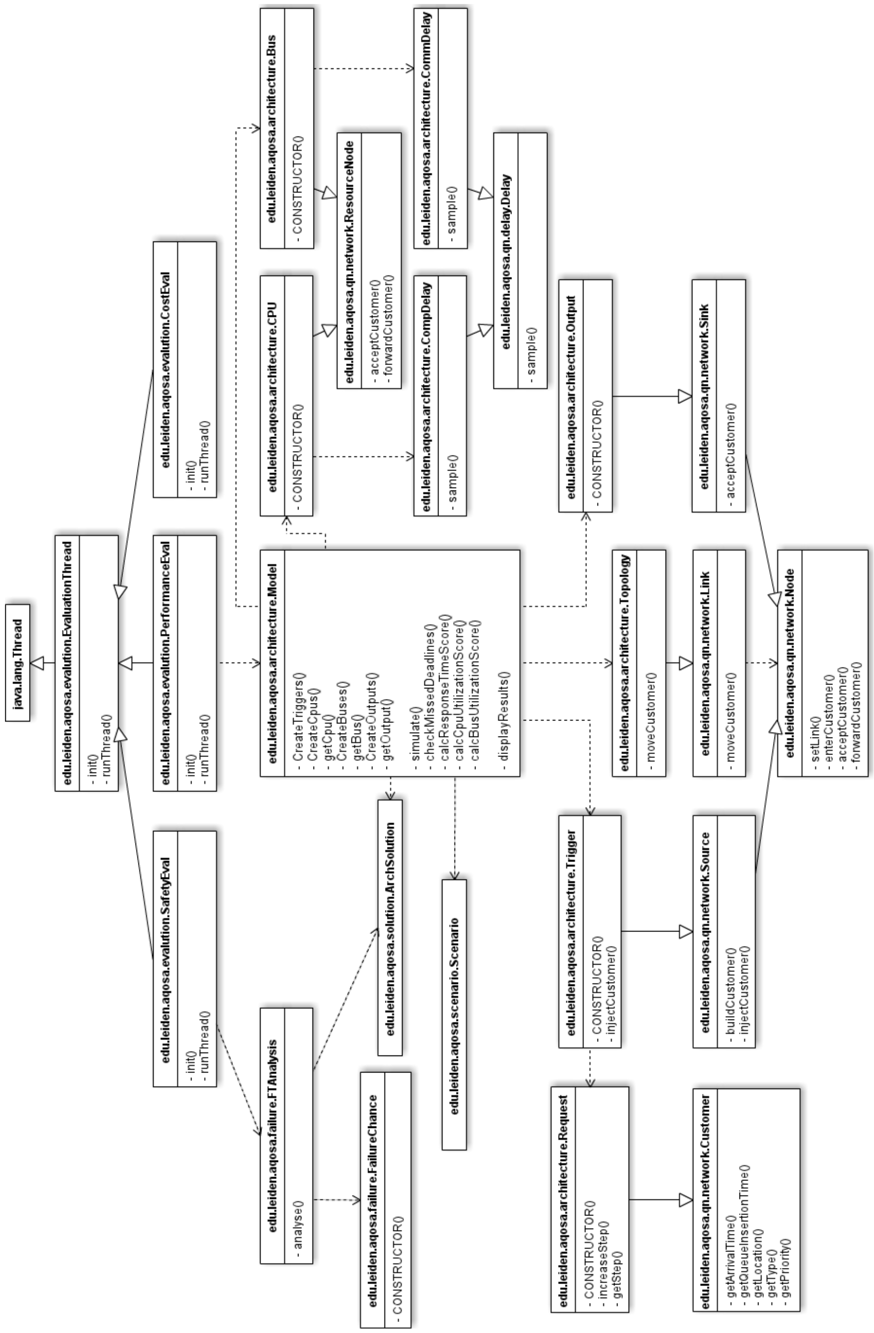

java.lang.Thread
edu.leiden.aqosa.evalution.EvaluationThread
- init()
- runThread()
edu.leiden.aqosa.evalution.CostEval
- init()
- runThread()
edu.leiden.aqosa.evalution.PerformanceEval
- init()
- runThread()
edu.leiden.aqosa.evalution.SafetyEval
- init()
- runThread()
edu.leiden.aqosa.architecture.Bus
- CONSTRUCTOR()
edu.leiden.aqosa.architecture.CPU
- CONSTRUCTOR()
edu.leiden.aqosa.qn.network.ResourceNode
- acceptCustomer()
- forwardCustomer()
edu.leiden.aqosa.architecture.CommDelay
- sample()
edu.leiden.aqosa.architecture.CompDelay
- sample()
edu.leiden.aqosa.qn.delay.Delay
- sample()
edu.leiden.aqosa.architecture.Output
- CONSTRUCTOR()
edu.leiden.aqosa.qn.network.Sink
- acceptCustomer()
edu.leiden.aqosa.architecture.Model
- CreateTriggers()
- CreateCpus()
- getCpu()
- CreateBuses()
- getBus()
- CreateOutputs()
- getOutput()
- simulate()
- checkMissedDeadlines()
- calcResponseTimeScore()
- calcCpuUtilizationScore()
- calcBusUtilizationScore()
- displayResults()
edu.leiden.aqosa.architecture.Topology
- moveCustomer()
edu.leiden.aqosa.qn.network.Link
- moveCustomer()
edu.leiden.aqosa.qn.network.Node
- setLink()
- enterCustomer()
- acceptCustomer()
- forwardCustomer()
edu.leiden.aqosa.solution.ArchSolution
edu.leiden.aqosa.scenario.Scenario
edu.leiden.aqosa.architecture.Trigger
- CONSTRUCTOR()
- injectCustomer()
edu.leiden.aqosa.qn.network.Source
- buildCustomer()
- injectCustomer()
edu.leiden.aqosa.failure.FTAnalysis
- analyse()
edu.leiden.aqosa.failure.FailureChance
- CONSTRUCTOR()
edu.leiden.aqosa.architecture.Request
- CONSTRUCTOR()
- increaseStep()
- getStep()
edu.leiden.aqosa.qn.network.Customer
- getArrivalTime()
- getQueueInsertionTime()
- getLocation()
- getType()
- getPriority()

Figure 4.12: *Evaluator module class diagram*

1. captures the processor types from AQOSA IR,
2. reads processor utilization values from queuing network simulation results, and transform them into an evaluation model.

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

4.6.3 Evaluator Module Implementation

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

4.7 Pareto front Monitoring

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

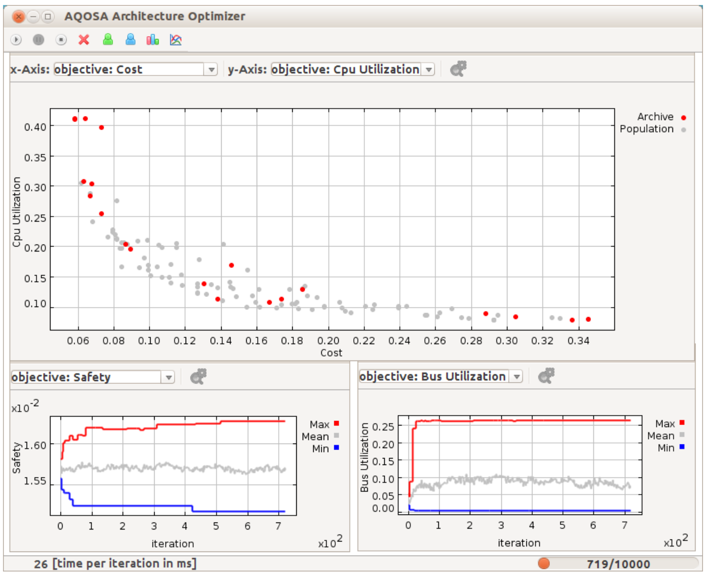


Figure 4.13: *AQOSA live Pareto front monitoring screenshot*

Application-based Interface

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

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

Web-based Interface

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

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

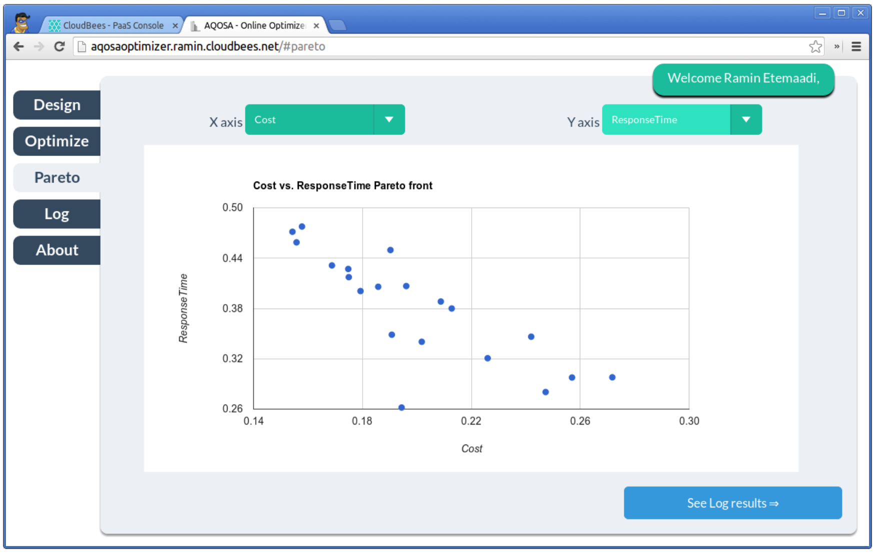


Figure 4.14: *AQOSA web-based interface screenshot*

4.8 Framework Algorithm Complexity

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

Table 4.2: *Algorithm complexity parameters*

Parameter	*Description*
d	the number of objectives (dimensions)
μ	the number of parents
λ	the number of offsprings
T	the length of simulation
n	the number of queues (number of nodes + number of buses) in the queuing network
p	the average number of events in the queue at times before the computed event

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

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

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

$$O(\frac{\mu}{\lambda} \cdot \log^{(d-1)}(\mu + \lambda) + T \cdot n \cdot p) \tag{4.1}$$

4.9 Summary

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

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

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

- It is modelling language independent. It can interoperate with various architectural modelling languages.
- It supports multiple degrees of freedom for automatically generating alternative architectures.
- It optimizes multiple quality attributes at once. To the best of our knowledge it is the first approach which supports evaluation and optimization of five quality attributes at the same time.

Chapter 5

Case Studies

In this chapter we validate the AQOSA framework, by applying it to three of software architecture design problems. These cases show the effectiveness and usefulness of automated quality-driven approach for the problem of software architecture design. These case studies will be the basis for the experiments in the following chapters, as well. By executing these experiments based on real-world industrial case studies, this chapter tries to address **RQ1** which is mentioned in Section 1.2:

> Can meta-heuristic optimization improve the process of designing efficient architectures for a set of given quality attributes in an industrial domain?

To the best of our knowledge this is the largest case study on architecture optimization of real industrial embedded software system.

This chapter is structured as follows. First, Section 5.1 demonstrates the AQOSA framework on a 3-objective optimization problem from the field of business information systems. Through this optimization experiment we also compare the state-of-the-art multi-objective optimization algorithms that are supported by AQOSA. Secondly, Section 5.2 presents a simple embedded software system for cruise control. This case has been used as a demonstration case in the field of multi-objective software architecture design optimization. This case study has 6 software components and between 1 and 6 hardware components. In contrast to the first case, here we study the optimization of 5 design objectives. Thirdly, Section 5.3 presents a study of a real industrial system: the instrument panel. This case was obtained from SAAB automotive. This case has 18 software components and between 1 and 18 hardware components. We study optimization of 5 design objectives. To the best of our knowledge, this is the largest case study on software architecture optimization – esp. of an industrial case. Together these cases demonstrate the effectiveness of the AQOSA framework. Section 5.4 summarizes and reflects on the findings of these experiments in this chapter.

5.1 Case Study 1: Business Report System

The goals of this first case study are (i) to demonstrate the effectiveness of the multi-objective optimization technique for the software architecture domain, and (ii) the comparison between some popular existing evolutionary multi-objective optimization algorithms. The algorithms that we compare are: NSGA-II [DAPM02], SPEA2 [ZLT02], SMS-EMOA [BNE07] and random search (for the domain of software architecture design).

The so-called business reporting system (BRS) is a system which lets users retrieve reports and statistical data about running business processes from a data base. The original case is described in [MKBR10]. It is loosely based on a real system [WW04].

5.1.1 Business Report System Components

From a component-based software development point of view, BRS is a 4-tier system consisting of 8 software components. The `WebServer` component handles user

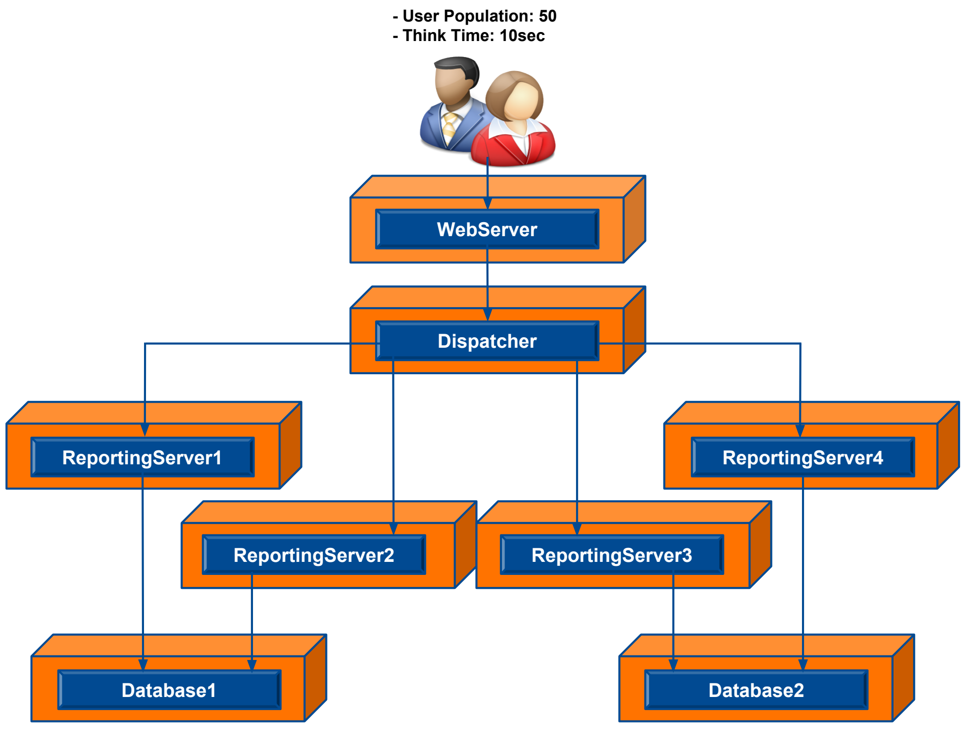


Figure 5.1: *BRS 4-tier components*

requests for generating reports or viewing the plain data logged by the system. It delegates these requests to a `Dispatcher` component, which in turn distributes the requests to four replicated `ReportingServers`. The replication helps balancing the load in the system, because the processing load for generating reports from the database contents is considered significant. The `ReportingServers` access two replicated `Databases` for the business data. Figure 5.1 depicts an overview of this system.

5.1.2 Execution Scenarios

In addition to a static view of the system, a behavioural view is needed in order to analyse dynamic quality properties of the architecture such as performance. Like the original case in [MKBR10], in this case study we also assume that in the usage scenario, 50 users concurrently access the system. Each user requests a report from the system and then looks at the results for 10 seconds before issuing the next request.

5.1.3 Experiment Setup

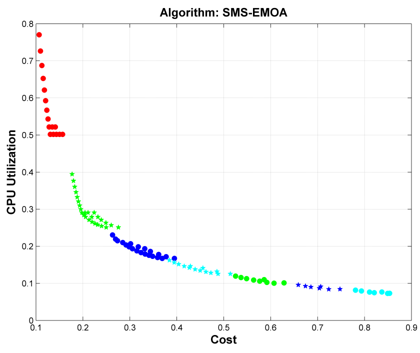
In the experiment for the BRS design problem, the following settings are chosen: initial population size: 100, parent population size: 100, number of offspring is 100, for SMS-EMOA $(100+1)$, reference point: $(1,1,1)$, archive size: 100, the number of generations: 500, crossover rate is set: 0.95, and constant mutation probability: 0.01. AQOSA was run 15 times ($\approx$ 3 hours per run) for each of the evolutionary multi-objective algorithms (NSGA-II, SPEA2, and SMS-EMOA), and also for random search. It was run on a computer with a 2-core processor (each core runs 3.16GHz and 6MB cache) and 2GB memory, using Java platform version 6.

5.1.4 Experiment Results

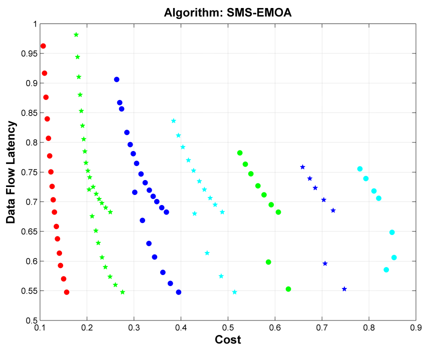
The resulting set of optimal solutions is visualized in a 3D Pareto front with respect to objectives response time, CPU utilization, and in Figure 5.3a. An interesting finding is that the resulting Pareto front is partitioned into several segments (7 segments in this typical run, in Figure 5.3a). By identifying (using different colors for different topologies) and mapping each individual from the set of solutions to the corresponding design architecture topology, it is observable that solutions from same segmentation share the same architectural topology: Solutions from the same segmentation share the same number of processor nodes. This could be the result of discontinuities in the search space caused by structural transitions.

For comparison between random search and evolutionary algorithms, in Figure 5.3b a 3D plot of random search is also presented. As can be seen, fewer segments (5 in this typical run) have been found by random search. In Figures 5.2a and 5.2b two different

Figure 5.2: *2D Pareto front comparison for Business Report system*

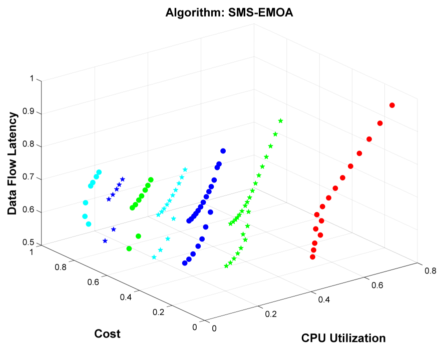


(a) *Cost vs. CPU utilization Pareto front*

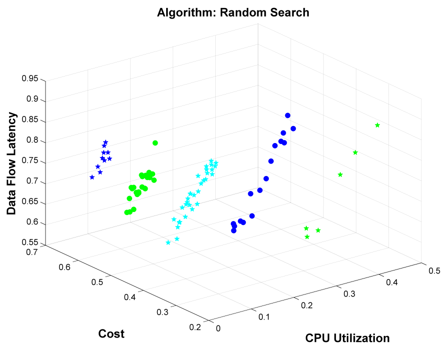


(b) *Cost vs. Response time Pareto front*

Figure 5.3: *3D Pareto front comparison for Business Report system*



(a) *3D Pareto front for SMS-EMOA algorithm*



(b) *3D Pareto front for random search*

Pareto fronts of two quality attributes (cost vs. CPU utilization and cost vs. response time) are depicted.

The boxplots of the hypervolume indicator with reference point $(1, 1, 1)^T$ for solution sets for NSGA-II, SPEA2, SMS-EMOA, and random search (for a set of 15 runs) are presented in Figure 5.4. From the boxplot (Figure 5.4), it can be seen that SMS-EMOA shows slightly better performance compared to the other algorithms. Random search (even with unbounded archive size) shows the worst performance by far.

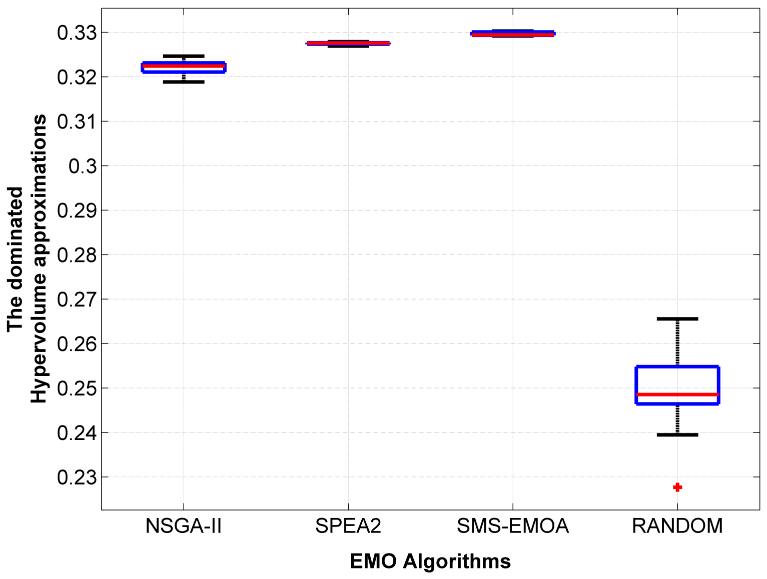


Figure 5.4: *The boxplots of the hypervolume indicator over 15 runs*

5.2 Case Study 2: Cruise Control System

This section discusses a case study about a cruise control (CC) system. This is a safety critical embedded systems from the automotive domain. Our case has been derived from the Vehicle Control system described by Gonzalez-Huerta [GH11]. It is loosely based on Control system described by Hudak et al. [HF07]. The goal of the cruise control system is to maintain a constant speed which is set by the driver. Practically, the system maintains a speed within some (small) interval around the set target speed.

5.2.1 Cruise Control Components

The cruise control system maintains the vehicle speed at the pre-determined value (target value) by storing the speed of the wheel rotation when the speed value is set and attempts to keep the throttle actuator at a position to maintain the vehicle speed at the target value. As the road inclination changes, the vehicle speed changes, and the throttle position should change to maintain the vehicle speed. The control system observes the speed difference between the current speed and the target value and either decreases or increases the throttle actuator position to counter act the speed differential. The algorithm to accomplish this is based on control theory.

Figure 5.5 depicts the flow of the data in CC system in Use Case Maps notation [ALBG99]. The system contains the following six components that aim to generate four output signals:

- `IN_CONTROL` Component: This component controls the input signals. It receives the brake pedal status and engine status which are its most important signals. It also receives the driver speed panel signals, to enable him to control the car speed. It calculates its output based on the values of input signals.
- `DESIRE_SPEED` Component: This component computes the desired speed based on the current car speed and signals from controller. It reads the selected speed and selected distance signals and generates the desired speed signal for the next 4 components by applying the control law.
- `AIRBAG` Component: This component generates airbag settings. It is responsible for sending proper airbag signals in case of collision.
- `SECURITY BELT` Component: In case of hard braking this component should send signals to the security belts. It receives data from two sensors (i) the obstacle sensor which is responsible for detecting obstacles on the road, and (ii) the distance sensor which is responsible for calculating distances to the next car.
- `THROTTLE` Component: This component is responsible for configuring the settings of the car throttle. It converts the relative speed into a throttle setting for the throttle actuator.
- `BRAKE` Component: This component should activate the car brakes when needed. This component only works in case of decreasing speed. This component converts the relative speed into a brake setting for the brake actuator.

5.2.2 Execution Scenarios

For this experimental study, we use the following scenario: a 2-second ($2000ms$) scenario is defined. At $t = 1900ms$, the obstacle sensor recognizes an obstacle in front of the car. Then, the cruise control system should react to this event within the predefined deadlines. Prior to this major event, the following periodic events happen continuously:

1. Every $50ms$: a signal is sent for security belt settings,
2. Every $100ms$: a signal is sent for airbag settings,
3. Every $150ms$: a signal is sent for throttle and brake settings.

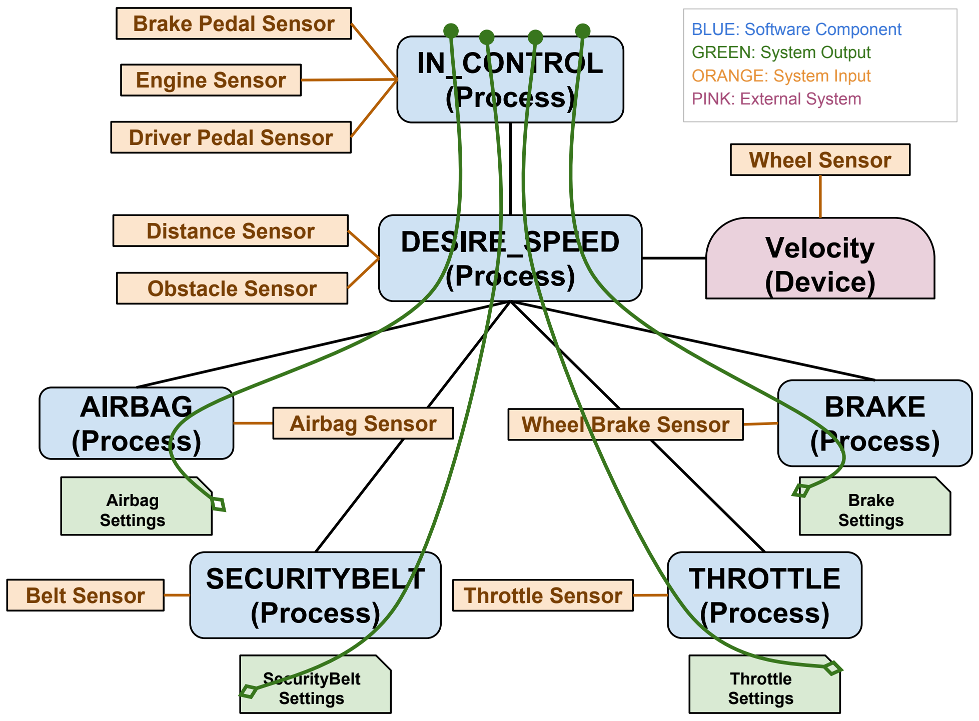


Figure 5.5: *Use Case Maps notation of Cruise Control system*

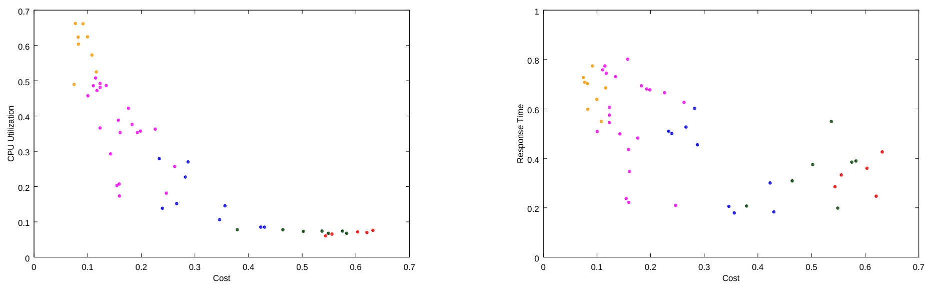
The system should generate 4 signals on its outputs. Based on the system requirements, these are the system deadlines:

- 50ms for security belt signal output,
- 80ms for airbag signal output,
- 30ms for brake signal output,
- 100ms for throttle signal output.

5.2.3 Experiment Setup

For solving the CC system design problem, a repository which consists of the following hardware components, has been considered:

Figure 5.6: *2D Pareto front comparison for cruise control system*



(a) *Cost vs. CPU utilization Pareto front* **(b)** *Cost vs. Response time Pareto front*

- 28 Processors types: ranging over 14 various processing speeds from 66MHz to 500MHz. Each of these has two levels of energy consumption. A processor is more expensive if it has a smaller chance of failure.

- 15 Buses: ranging over five types of bandwidths (128, 160, 192, 224 and 256 kbps) and each type has three variants with different latencies (1, 3 and 5 ms). Again, a bus is more expensive if it supports higher bandwidth or smaller time delay.

The AQOSA optimization was run with the following parameter settings on a computer with a 2-core processor (each core runs 3.16GHz and 6MB cache) and 2GB memory: initial population size (α) = 100, parent population size (μ) = 50, number of offspring (λ) = 50, archive size (with crowding type) = 50, number of generations = 10000, crossover rate is set to 0.95.

5.2.4 Experiment Results

The 2D Pareto fronts in Figure 5.6 show the trade-off between objectives. Improving one dimension often implies a decrease of the other dimension. Different colors in the plots correspond to different numbers of nodes in the architecture: black for 1-node, orange for 2-node, magenta for 3-node, blue for 4-node, green for 5-node and red for 6-node architectures. As can be seen there is no black node which means AQOSA could not find an optimal solution with a 1-node architecture. As can be seen, the results of this case study is not clearly segmented like the previous case study, but still similar architectures are close to each other.

In Figure 5.7 a 3D Pareto front of a set of optimal solutions is depicted. It shows the trade-off between Cost vs. CPU utilization vs. Response time. The colors in the plot are the same as the color in the 2D plots.

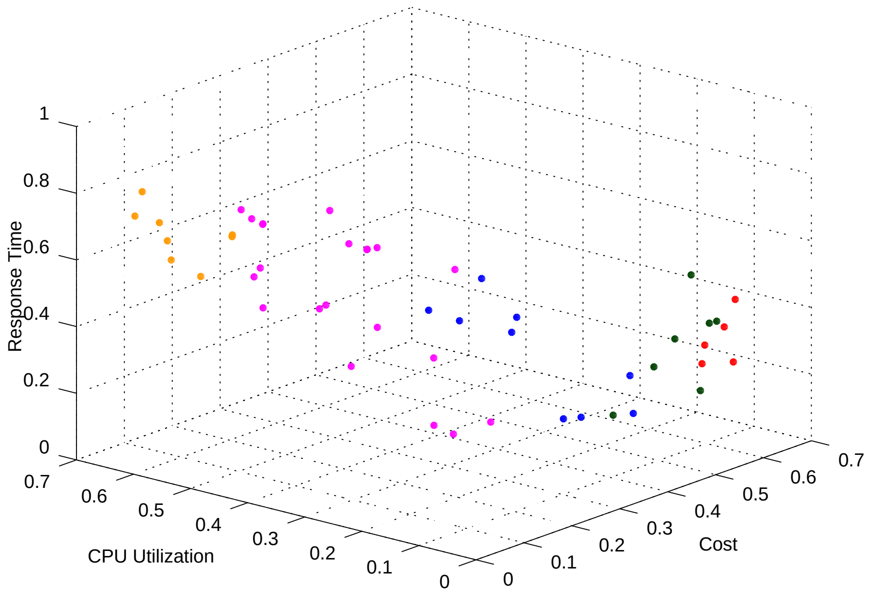


Figure 5.7: *3D Pareto front for cruise control system*

5.3 Case Study 3: SAAB Instrument Cluster Sub-System

The goal of this case study is to explore in what aspects meta-heuristic optimization improves the process of designing architectures for a set of given quality attributes in an industrial context. This real-world case study shows how an architecture optimization framework helps system architects make better decisions by complementing their domain knowledge and experience.

This study was conducted at Saab Automobile AB in order to evaluate and validate the AQOSA framework in a large industrial design problem. To enable the validation of the results, an existing realization for the Saab 9-5 Instrument Cluster Module ECU (Electronic Control Unit, a node in a network) and the surrounding sub-system have been selected. The aim of the optimization experiment is to find a solution that is cheaper than the current realization while fulfilling the same requirements and constraints. This constitutes a fair comparison between the solution proposed by AQOSA and the current industrial realization. This problem is an ever-existing problem in the automotive industry, because of high cost pressure and variation in constraints during the lifetime of a system design.

User function	*Description*
Vehicle Speed Indication	Shows the speed of the car on the speedometer gauge. The vehicle speed information is based on the wheel speed sensors, filtered and converted into the chosen metric (either km/h or miles/h).
Coolant Temperature Indication	Shows the temperature of the engine coolant on the coolant temperature gauge. The coolant temperature information is based on a temperature sensor, filtered and compensated to show the correct temperature.
Selected Gear Indication	Displays the current automatic gear position of the vehicle. (Possible gear positions are P, R, N, D, and L.) The information is based on position sensors in the Automatic Gearbox.
Engine Speed Indication	Shows the engine revolutions per minute information on the tachometer gauge. The engine speed information is based on the crankshaft sensor, and filtered before it is shown.
Odometer Indication	Shows the distance travelled by the car. The odometer is based on the wheel speed sensors, filtered and converted into distance in the chosen metric (either km or miles).
Ignition Switch Power Moding	The global system power mode in the car. The system power mode information is based on the ignition key switch, with the possible positions OFF, ACCESSORY, RUN, and CRANK. This information is transmitted to all parts of the system.
Outside Air Temperature Indication	Shows the outside temperature on the display. The outside temperature information is based on a temperature sensor, filtered and compensated to show the correct temperature.
Low Washer Indication	Notifies the driver when the windshield wiper fluid level is low. The notification is based on a fluid level switch.

Table 5.1: *Description of user functions for the SAAB Instrument Cluster sub-system*

5.3.1 Instrument Cluster Components

The purpose of the Instrument Cluster sub-system is to provide the driver with information that is required when driving the car. The information is processed by user functions based on sensor values, and presented on gauges or displays located in front of the driver. The user functions (and their input/output devices) included in the study are shown in Table 5.1.

User functions like the ones in Table 5.1 can be complex due to regulations, safety properties, variation between different car models, or other quality constraints. In

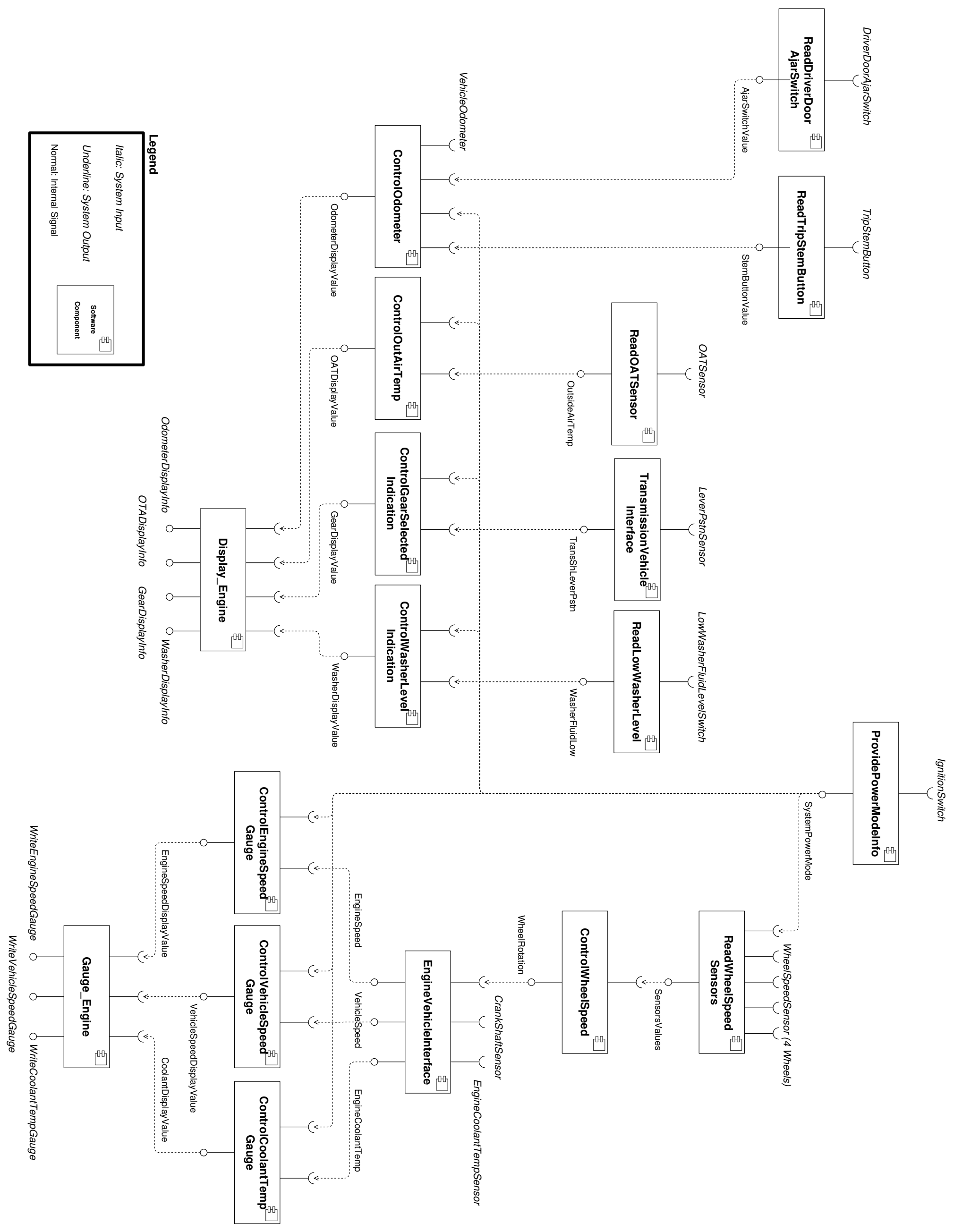


Figure 5.8: *Component diagram of SAAB Instrument Cluster sub-system*

general, user functions have to satisfy multiple requirements and quality constraints. User functions are specified and implemented using one or more software components. Figure 5.8 shows the component diagram of the Instrument Cluster sub-system. The diagram shows how the software components are connected to satisfy the user functions in Table 5.1. Figure 5.8 also illustrates the size and complexity of this case.

5.3.2 Execution Scenarios

The user functions in Table 5.1 are typically triggered by events captured by their input devices (sporadic tasks), or by an invocation from a scheduler within the sub-system (periodic tasks). These tasks were defined by describing changes in switch/sensor values. The sporadic tasks are defined in Table 5.2 and the periodic tasks are defined in Table 5.3.

As an example of a stimulus and system response for a sporadic task we will look at the first task in Table 5.2. The `DriverDoorAjarSwitch` detects that the driver door is opened while the vehicle is parked and the engine is `OFF`. Then the `Odometer` value shall be displayed within 500ms. The other tasks in Table 5.2 are of similar kind.

As an example of a stimulus and system response for a periodic task we will look at the first task in Table 5.3. The first periodic task is executed when the engine is running and the vehicle speed is changing. Then the `VehicleSpeed` signal shall be transmitted periodically each 100ms, the `VehicleSpeedDisplayValue` shall be calculated periodically each 100ms, and the Vehicle Speed pointer shall begin to move

Stimulus	***System response***
$IgnitionSwitch = 0$ (OFF) $\wedge$ $DoorAjarSwitch : False \rightarrow True$	Odometer shall be displayed within 500 ms
$IgnitionSwitch = 0$ (OFF) $\wedge$ $TripStemButton : False \rightarrow True$	Odometer shall be displayed within 500 ms
$IgnitionSwitch : 0$ (OFF) $\rightarrow 2$ (RUN)	The Engine Speed pointer shall begin to move to the correct position within 150 ms
$IgnitionSwitch : 0$ (OFF) $\rightarrow 2$ (RUN)	The Vehicle Speed pointer shall begin to move to the correct position within 150 ms
$IgnitionSwitch = 2$ (RUN) $\wedge$ $WasherFluidSensor : False \rightarrow True$	Low Washer Fluid Indicator shall be illuminated within 250 ms
$IgnitionSwitch = 2$ (RUN) $\wedge$ $GearLeverPositionSwitch : 1$ (P) $\rightarrow 4$ (D)	Driving gear position shall be displayed within 100 ms

Table 5.2: *Sporadic tasks included in SAAB Instrument Cluster sub-system*

Stimulus	*System response*
$\mathtt{IgnitionSwitch} = 2$ (RUN) $\wedge$ $\mathtt{VehicleSpeedSensor}:$ $0 \rightarrow 100\mathtt{km/h}$	`VehicleSpeed` signal shall be transmitted periodically each 100 ms. `VehicleSpeedDisplayValue` shall be calculated periodically each 100 ms. The Vehicle Speed pointer shall begin to move to the correct position within 150 ms.
$\mathtt{IgnitionSwitch} = 2$ (RUN) $\wedge$ $\mathtt{CrankshaftSensor}:$ $0 \rightarrow 5000\mathtt{rpm}$	`EngineSpeed` signal shall be transmitted periodically each 100 ms. `EngineSpeedDisplayValue` shall be calculated periodically each 100 ms. The Engine Speed pointer shall begin to move to the correct position within 150 ms.
$\mathtt{IgnitionSwitch} = 2$ (RUN) $\wedge$ $\mathtt{OutsideTempSensor}:$ $0 \rightarrow 100\%$	The outside air temperature shall be calculated once every second. The outside air temperature shall be displayed within 100 ms.
$\mathtt{IgnitionSwitch} = 2$ (RUN) $\wedge$ $\mathtt{CoolantTempSensor}:$ $0 \rightarrow 100\%$	`EngineCoolantTemp` signal shall be transmitted periodically each 100 ms. `CoolantDisplayValue` shall be calculated periodically each 100 ms. The Coolant Temp pointer shall begin to move to the correct position within 150 ms.

Table 5.3: *Periodic tasks included in SAAB Instrument Cluster sub-system*

to the correct position within $150\mathtt{ms}$. The other tasks in Table 5.3 are of similar kind.

As stated before, user functions like the ones in Table 5.1 are typically implemented by one or more software components. We use sequence diagrams to model the collaboration among these components. Sequence diagrams represent specific scenarios. The use of sequence diagrams was a convenient way of capturing the information needed for AQOSA. It required domain knowledge to create them, especially to understand how the components interact and how to add the timing constraints. Ten sequence diagrams are made (as shown in Figures 5.9 to 5.18), and the sequence diagrams contain between 3 and 6 software components with 4 components as the average.

These sequence diagrams need to be enhanced with information required for AQOSA quality attributes evaluation process. As such, AQOSA requires the number of cycles for executing each operation is needed. The number of execution cycles for each operation was obtained by analyzing the source code of the component with the SCoPE simulation framework [PHS+04]. Components for which the source code was not accessible, were analyzed manually by reading the requirement specification and comparing this to similar components, to estimate the number of execution cycles. This information is added next to the activation on the sequence diagram. AQOSA also requires the number of bytes being sent in order to compute communication loads. The number of bytes sent in each message was obtained by calculating the size of the data that is sent. This information is added below the messages.

Here, the first sequence diagram is discussed in details. The other sequence diagrams are similar, and follow the same principles. For example, as described in the first sequence diagram in Figure 5.9, the operation `CalculateOAT()` takes 2744 cycles to execute. This number of execution cycles for the `CalculateOAT()` operation was obtained by analyzing the source code of the `ControlOutAirTemp` component with the SCoPE simulation framework. Also in the same diagram (Figure 5.9), we see that the size of message `CalculateOAT()` is 1 byte.

In addition to the information above, AQOSA also needs the timing constraints described in Table 5.2 and Table 5.3. This information was added to the sequence diagram in a standard way to the left of the sequence diagram and with the help of notes. Again, an example from the first diagram is that on the left side in Figure 5.9, we see that the maximum delay for the whole scenario should be less than $100ms$. The note added to the message `ObtainOAT()` constrains it to be invoked each $1000ms$.

As can be seen in Figure 5.9, there are three actors (*User*, *OAT Sensor*, and *Display*) and three software components (`ReadOATSensor`, `ControlOutAirTemp`, and `Display_Engine`). After these sequence diagrams have been created, it is a simple task to feed this information into AQOSA framework.

Constraint	*Description*
1	Wheel Speed Sensor shall always be connected to the Brake Module
2	Crankshaft Sensor and Engine Coolant Temp Sensor shall always be connected to the Engine Module
3	The Brake Module and the Engine Module shall always be connected to a HS CAN bus

Table 5.4: *Deployment constraints for SAAB Instrument Cluster sub-system*

The user functions (see Table 5.1) and the timing constraints (see Table 5.2 and Table 5.3) are needed to obtain the required input to AQOSA, but there is one more important type of constraint for this domain. Deployment constraints state which software components may or may not be mapped onto whith hardware components. The deployment constraints used in our case study are stated in Table 5.4.

In the following, the sequence diagram for each user function of the system is depicted. It contains the deadline and the required number of cycles for each operation.

Outside Air Temperature Indication

Figure 5.9 depicts the sequence diagram of `OutsideAirTemperatureIndication`. As already mentioned, it consists of three actors (*User*, *OAT Sensor*, and *Display*) and three software components.

`ReadOATSensor` is being called every 100ms. `ReadOATSensor` itself gets data from the sensor and calls `ControlOutAirTemp`. After that, `ControlOutAirTemp` consequently calls `Display_Engine` to display proper information. The deadline for this task is 100ms.

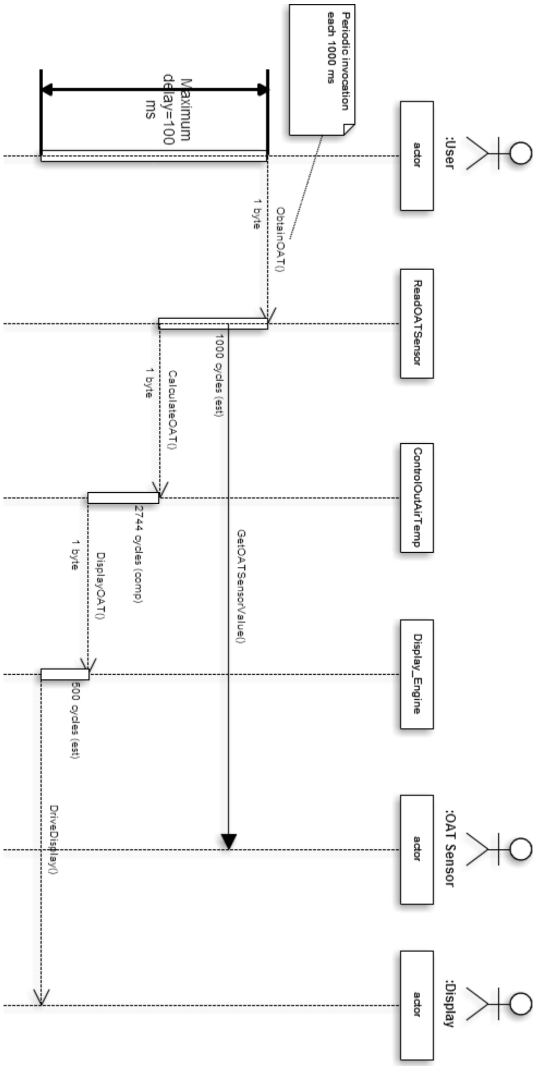


Figure 5.9: *Sequence diagram of* `OutsideAirTemperatureIndication` *user function*

Vehicle Speed Indication

Figure 5.10 depicts the sequence diagram of `VehicleSpeedIndication`. It is also a periodic task, in which `ReadWheelSpeedSensors` component is being called every 100ms. It gets needed data from the wheel sensors and calls the `ControlWheelSpeed` component. Then, `ControlWheelSpeed`, calls `EngineVehicleInterface`, and it calls `ControlVehicleSpeedGauge`, and it calls `Gauge_Engine` respectively. At the end, `Gauge_Engine` sends data to physical or electronic vehicle speed gauge. The deadline for this task is 50ms.

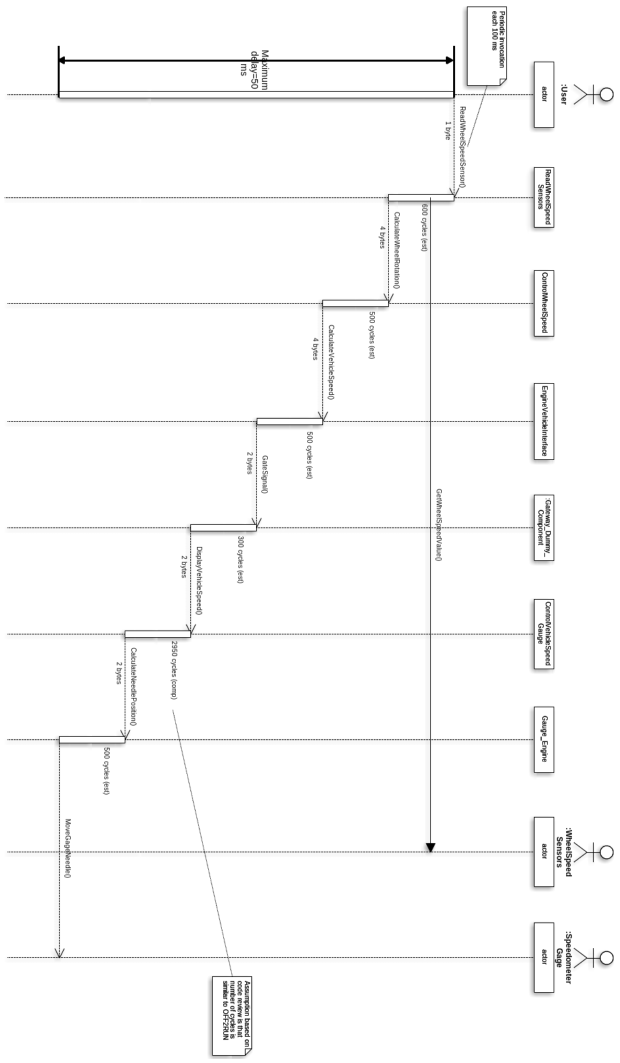


Figure 5.10: *Sequence diagram of* `VehicleSpeedIndication` *user function*

Vehicle Speed on Start

Figure 5.11 depicts the sequence diagram of `VehicleSpeed_onStart`. It is more or less similar to the previous one, but this task is a sporadic task, instead of a periodic one. Hence, it only happens when the driver turns the ignition switch. As can be seen from the diagram, the components interact with two sensors, *Ignition Switch*, and *Wheel Speed Sensors*. The interactions between the components are quite similar to the previous one. However, the deadline for this task is 150ms.

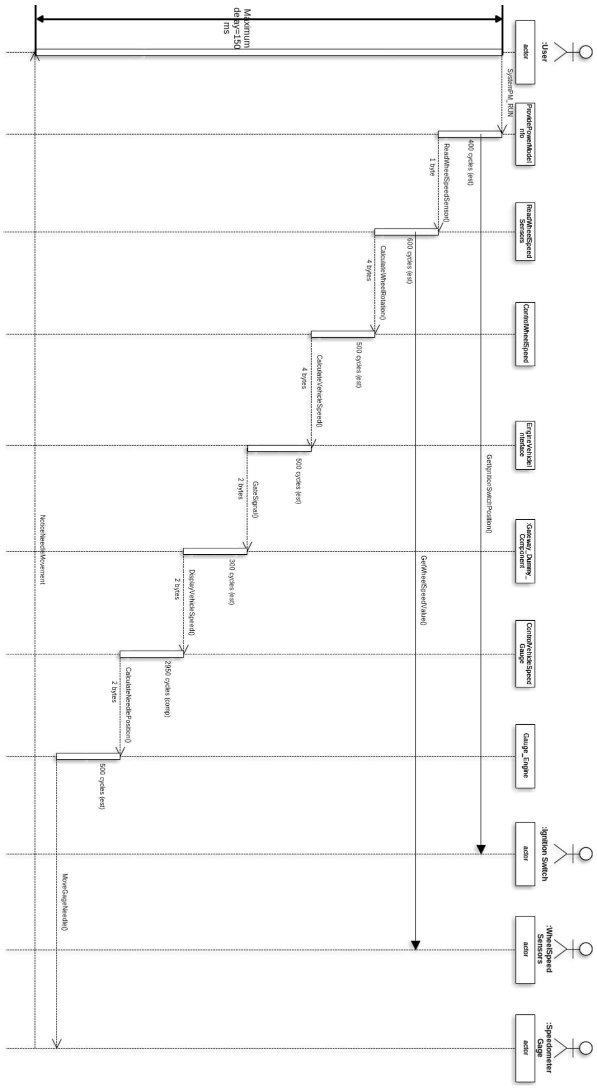


Figure 5.11: *Sequence diagram of `VehicleSpeed_onStart` user function*

Engine Speed Indication

The sequence diagram for `EngineSpeedIndication` is depicted in Figure 5.12. It is a periodic task. It consists of of three actors (*User*, *Crankshaft Sensor*, and *Engine Speed Gauge*) and three software components. At the start, the `EngineVehicleInterface` component is being called every 100ms. It gets needed data from the crankshaft sensor and calls the `ControlEngineSpeedGauge` component. It also adjusts the data and calls the `Gauge_Engine` component. After that, `Gauge_Engine` sends data to physical or electronic engine speed gauge. The deadline for this task is 50ms.

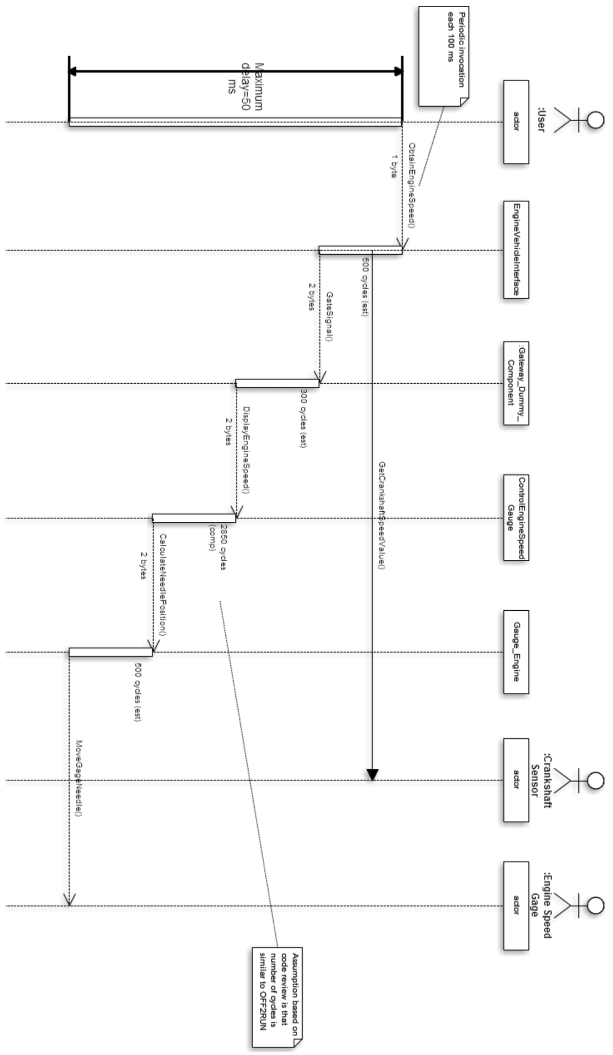


Figure 5.12: *Sequence diagram of* `EngineSpeedIndication` *user function*

Engine Speed on Start

The sequence diagram for `EngineSpeed_onStart` is depicted in Figure 5.13. Again, it is quite similar to the previous one, but it is a sporadic task. Therefore, we can see *Ignition Switch* sensor is involved in the sequence. The deadline for this task is 150ms.

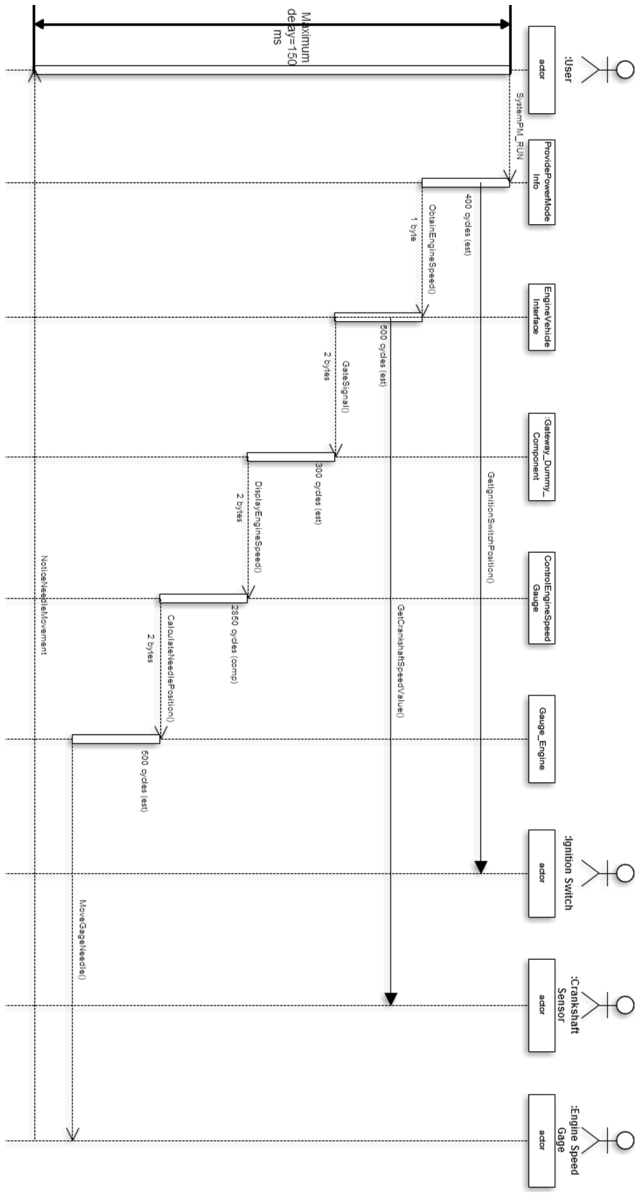


Figure 5.13: *Sequence diagram of* `EngineSpeed_onStart` *user function*

Coolant Temperature Indication

`CoolantTemperatureIndication` is another periodic task. Figure 5.14 shows the sequence diagram for it. As shown in the diagram, the `EngineVehicleInterface` component is being called every 100ms. It gets the required data for this task from the *Engine Coolant Temperature Sensor* and calls the `ControlCoolantTempGauge` component to adjust the data. Then, `ControlCoolantTempGauge` calls `Gauge_Engine`, and finally, it sends data to the physical needle. The system has 150ms time to finish this task.

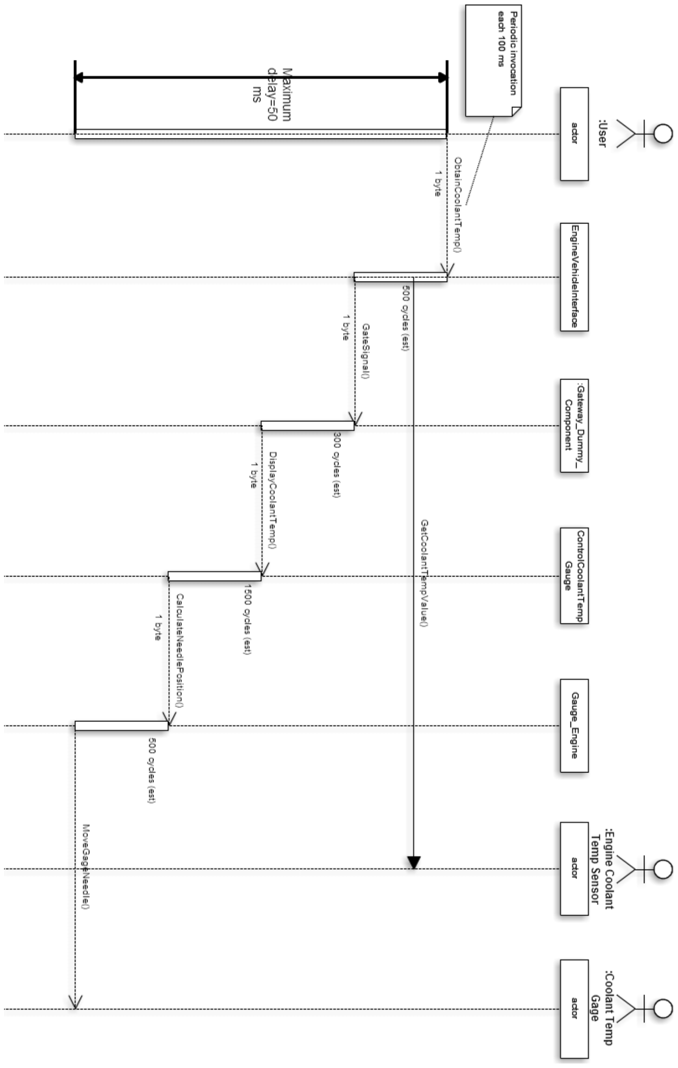


Figure 5.14: *Sequence diagram of* `CoolantTemperatureIndication` *user function*

Selected Gear Indication

The sequence diagram of `SelectedGearIndication` is shown in Figure 5.15. This task is a sporadic task and it is being triggered by *Lever Position Sensor*. When this event happens, `TransmissionVehicleInterface` will be called. It sends the data for the `ControlGearSelectedIndication` component to interpret the data. After that, it manipulates the data and sends it to `Display_Engine` to be displayed for the *User*. The deadline for this task is 100ms.

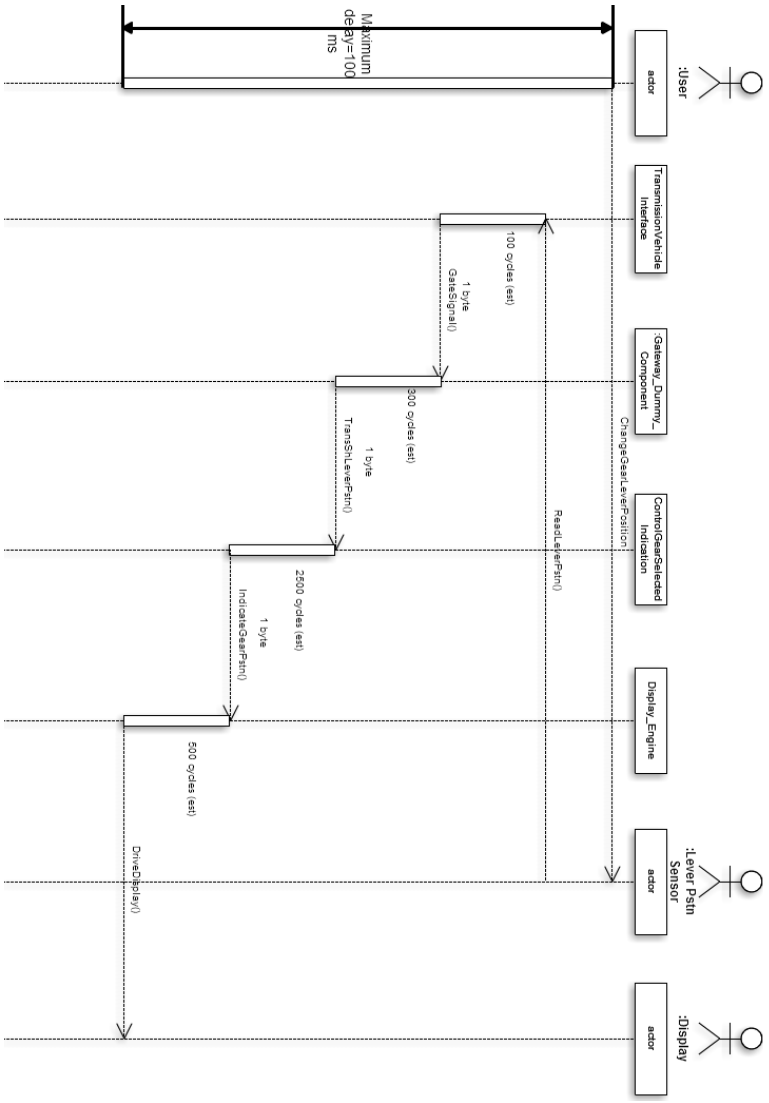


Figure 5.15: *Sequence diagram of* `SelectedGearIndication` *user function*

Odometer Indication on Trip Stem Button Activation

This is a sporadic task which will be run when the user pushes the *Trip Stem Button*. Figure 5.16 depicts the sequence diagram of it. It interacts with an external system named *Control Odometer Storage* which stores odometer data for the future retrieval purposes. As can be observed from the diagram, after the trigger event happens, the `ReadTripStemButton` component is called. It calls the `ControlOdometer` component. This component interacts with an external system and then sends proper data to `Display_Engine` to be displayed on the dashboard panel. The deadline for this task is 500ms.

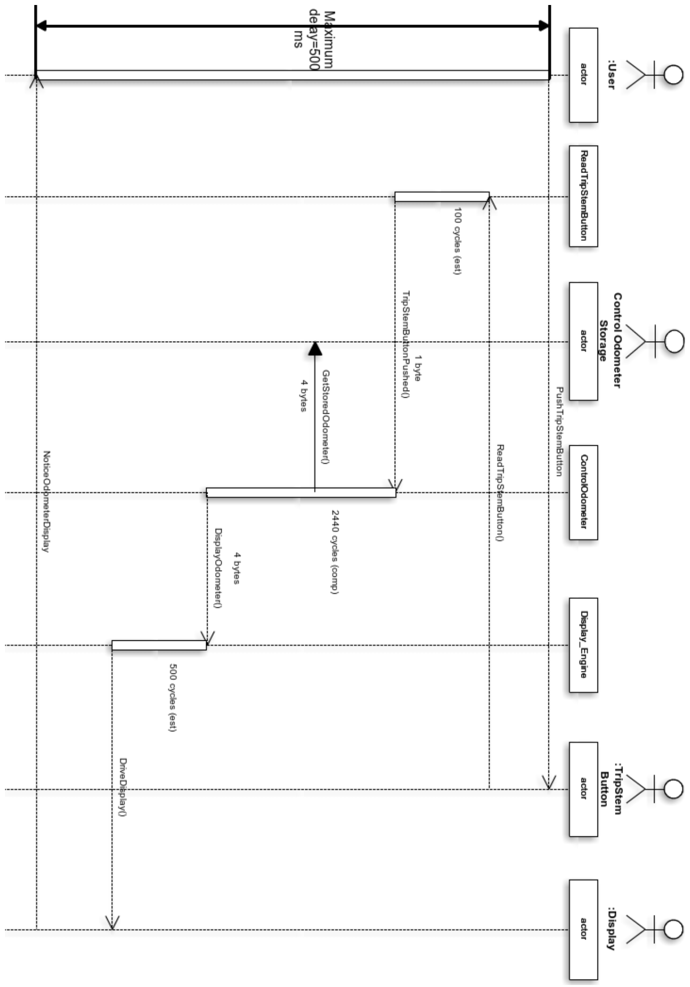


Figure 5.16: *Sequence diagram of* `OdometerIndication` *user function on Trip Stem Button Activation event*

Odometer Indication on Door Ajar Activation

This is a sporadic task is again quite similar to the previous one, however it will be run when the driver opens the car's door. Figure 5.17 depicts the sequence diagram of this task. It also interacts with the external system of *Control Odometer Storage* and its deadline is 500ms. The main sequence of data and components calling is the same as previous one.

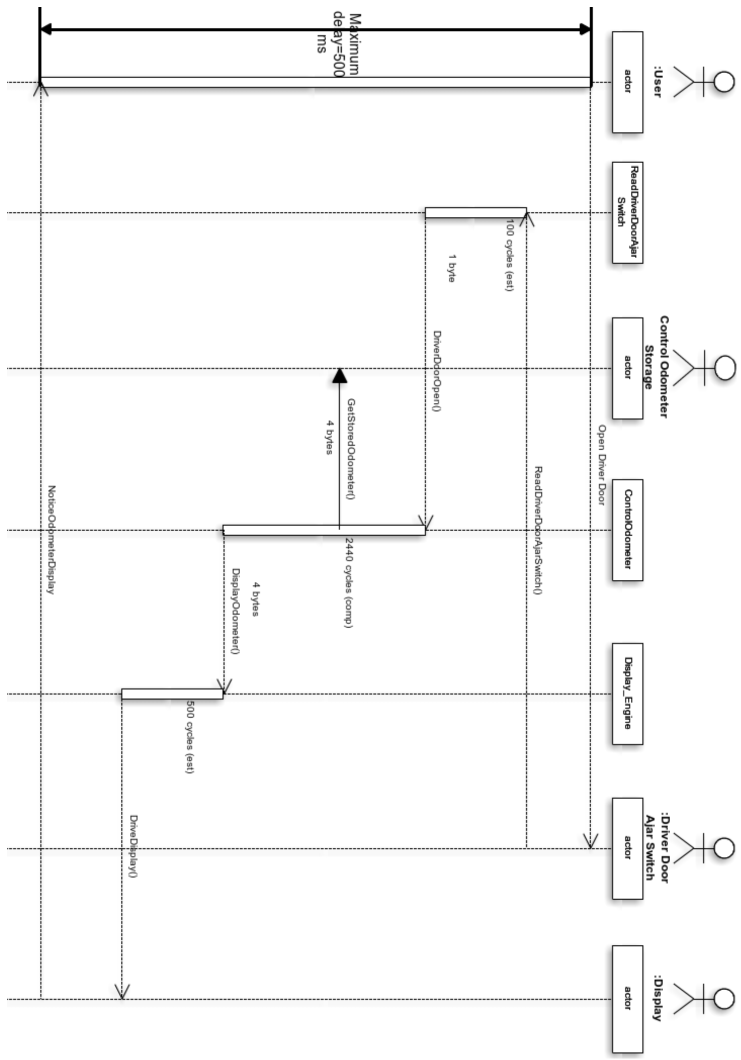


Figure 5.17: *Sequence diagram of* `OdometerIndication` *user function on Door Ajar Activation event*

Low Washer Indication

The sequence diagram for `LowWasherIndication` is depicted in Figure 5.18. Its sequence is pretty straight forward. The `ReadLowWasherLevel` component is called when the switch detects that the washer fluid is at a low level. `ReadLowWasherLevel` gets data from the sensor and calls the `ControlWasherLevelIndication` component. It adjusts the data and sends proper information to `Display_Engine` to be displayed on the dashboard panel.

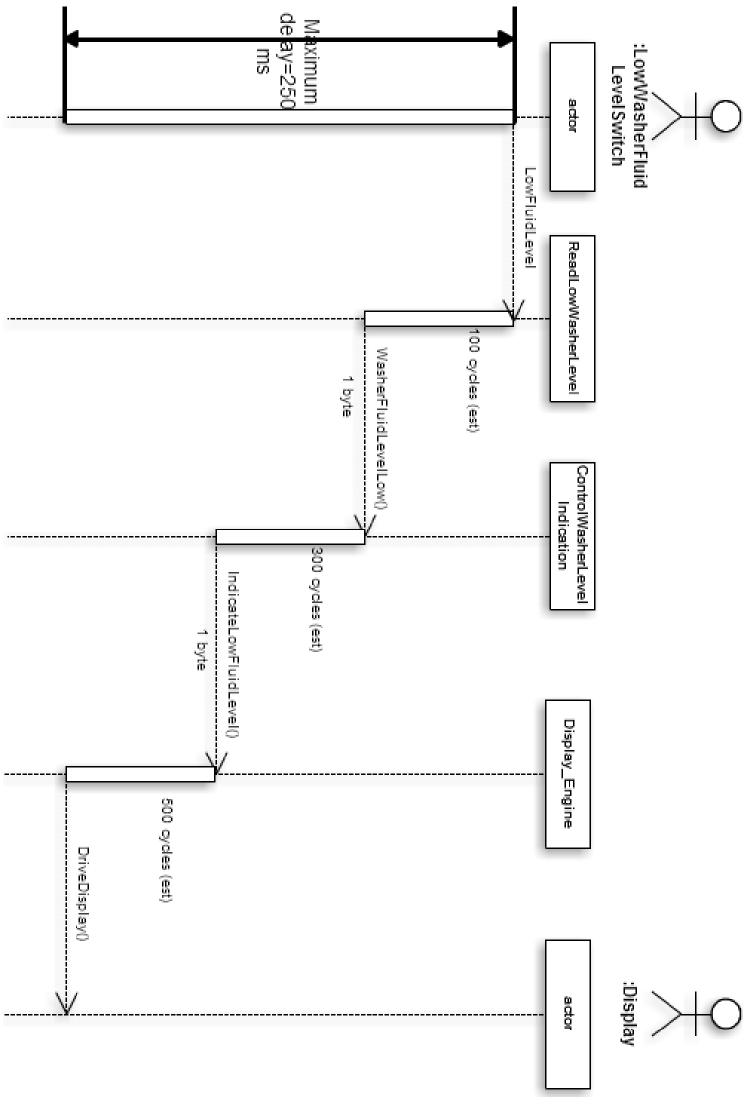


Figure 5.18: *Sequence diagram of* `LowWasherIndication` *user function*

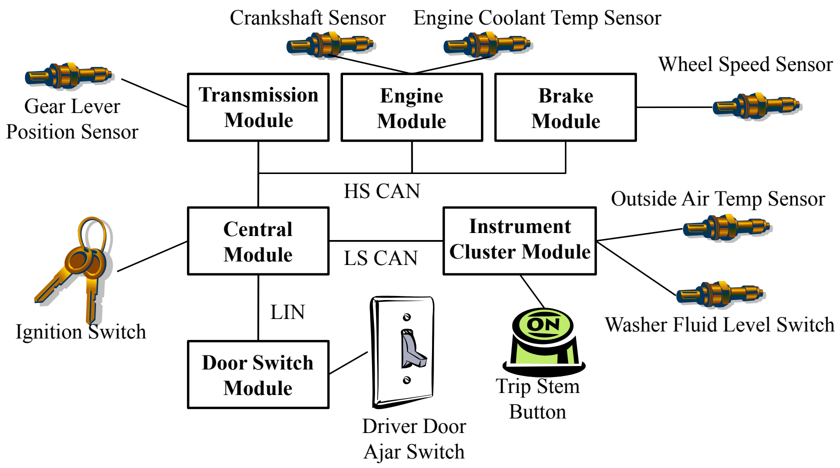


Figure 5.19: *Current realization of SAAB Instrument Cluster sub-system*

5.3.3 Current Realization

The task for AQOSA is to propose candidate software architectures that are optimized with regards to five quality attributes: bus utilization, cost, CPU utilization, response time, and safety. From the proposed realizations, one solution shall be selected based on the quality attribute values and deployment constraints. The current realization of the Instrument Cluster sub-system is shown in Figure 5.19. This realization is used as baseline when validating the results of the study. In Figure 5.19 each box is an ECU-processor, and the lines between these boxes are communication buses. The displays and gauges are part of the Instrument Cluster Module (To be consistent with Saab terminology, the term *'module'* has been used for a hardware node in this case study).

The hardware data and cost of the current realization will be used as a baseline for evaluating the proposed solutions. The most important hardware data and the hardware cost are shown in Table 5.5.

5.3.4 Experiment Setup

For generating new software architectures, a repository of hardware components was assembled based on predictions of planned system performance and anticipated market prices;

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

ECU	*Cost (USD)*	*MIPS*
Brake Module	100	80
Central Module	50	60
Door Switch Module	15	10
Engine Module	120	100
Instrument Cluster Module	50 [1]	60
Transmission Module	50	40
Piece of hardware	***Cost (USD)***	***kbps***
HS CAN (cost/module)	1	500
LS CAN (cost/module)	0,25	33
LIN (cost/module)	0,1	10

Table 5.5: *Hardware data and cost*

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

As an estimate of the size of the design space of this case study, consider the following reasoning: Assume we fix that the architecture has six processors (like the current realization, see Figure 5.19, and thus exclude many alternatives in hardware topology) and three bus lines for their interconnections. For these constraints there are $10^6 \cdot 4^3$ different possibilities, which is equal to $64 \cdot 10^6$ architectures. When also considering variations in the architecture topologies, this number would be considerably higher. This design space needs to be searched in an efficient manner.

After defining the above hardware options, AQOSA was run 30 times using the NSGA-II algorithm with the following parameter settings: initial population size(α)=2000, parent population size (μ)=100, number of offspring(λ)=50, archive size=50, number of generations=5000, crossover rate is set to 0.75, constant mutation probability is 0.01 and all quality attributes are aimed to be minimized.

5.3.5 Experiment Results

Figure 5.20, Figure 5.21, and Figure 5.22 show the results produced by AQOSA for the Instrument Cluster sub-system. Figure 5.20 contains a set of 2D plots for all pairs of two out of the five quality attributes that are considered in this design problem. Hence these depict 2-dimensional views on the resulting 5-dimensional Pareto front. Each box shows a particular view of the 5D result and the relation between two attributes.

[1]The design cost driven by "look and feel" requirements for display panel have been excluded because this is a cost that is common to all solutions.

Some of the attributes are in conflict like CPU util and cost. Some of the attributes are positively related, like response time and CPU util, and finally, some of them are independent such as bus util and safety or response time and safety. Pareto fronts are used as support for trade-offs between attributes by showing which solutions that are the best, i.e. Pareto-optimal, and also showing which trade-offs are possible. Different

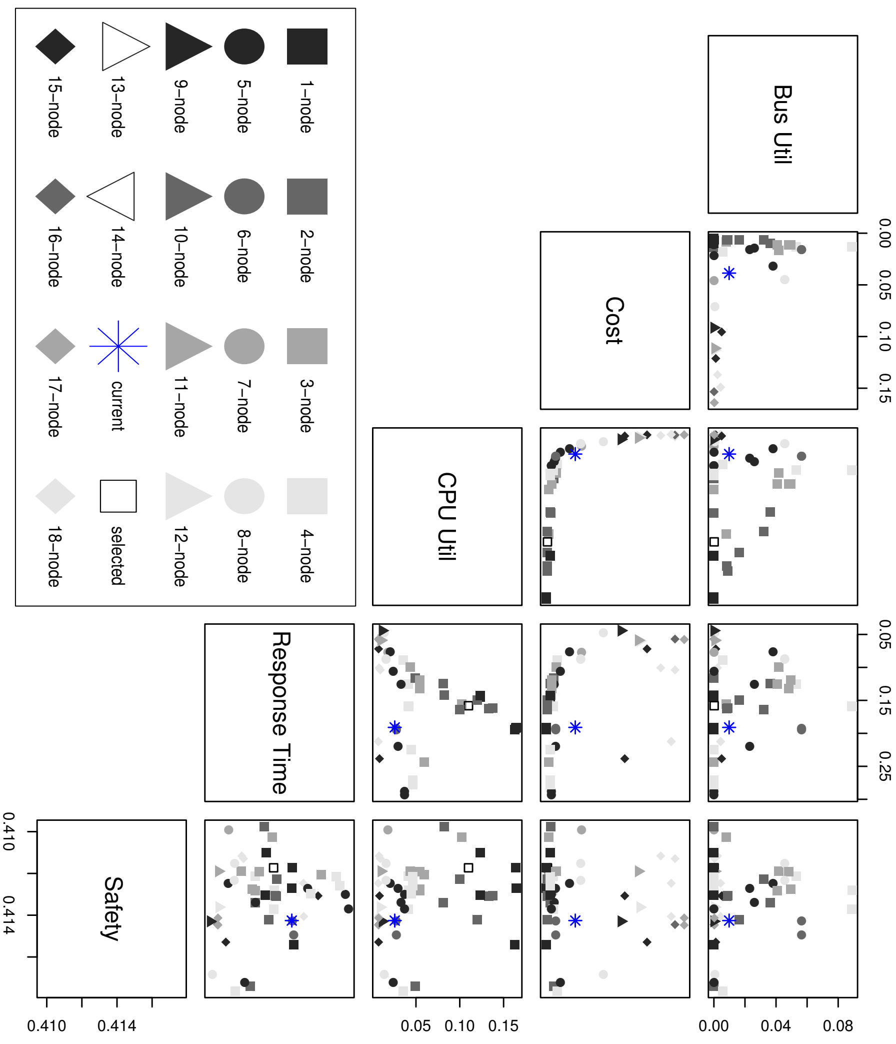


Figure 5.20: *Pareto front views for all pairs of two out of five quality attributes*

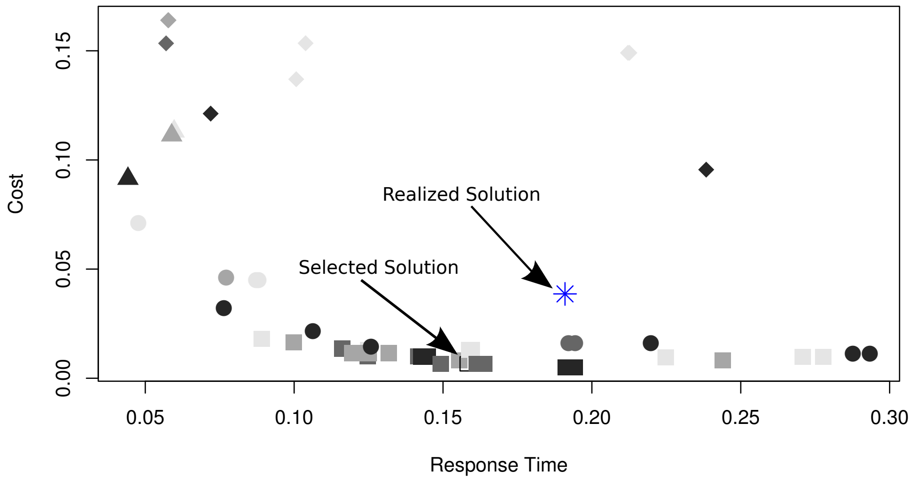


Figure 5.21: *Response time vs. cost Pareto front*

shapes and shades of gray in the plot show solutions with different numbers of nodes in the architecture: from black square for 1-node, dark gray square for 2-node, gray square for 3-node, light gray square for 4-node, black circle for 5-node, dark gray circle for 6-node, and continues up to light gray diamond for 18-node architectures. As can be seen there are architecture solutions containing 1-, 2-, 3-, 4-, 5-, 6-nodes, and more nodes up to 18. In Figure 5.20 the "$*$" sign shows the current realization (see Figure 5.19). Figure 5.21 shows a zoom-in on the Pareto front of Response time vs. Cost. The Pareto fronts in Figure 5.20 illustrate that the current realization is not Pareto-optimal. For example, Figure 5.21 shows that AQOSA found other solutions that are better both in response time and cost.

Figure 5.22 shows the parallel coordinate plot of the optimized solutions. In this plot each line (from left to right across the entire diagram) represents one architecture solution: the (normalized) values for the different attributes of the solution are marked by the crossing of this line and the vertical attribute axes. From this representation of the data, it can be observed that no architecture solution is optimal in all attributes.

Next, we discuss some of the proposed solutions in more detail. The cheapest candidate contains 1 node with a 60 MIPS processor, no communication bus, and a total cost of 50 USD. The safest candidate contains 7 nodes connected by 8 buses with a total cost of 459 USD. Table 5.6 shows the attribute values for these candidate solutions. The current realization (see Figure 5.19) is included as for comparison. In Table 5.6, bus utilization is shown as average value (1.0 represents maximum

Candidate	*Bus Util*	*Cost*	*CPU Util*	*Response time*	*Safety*
Cheapest	0	0.050	0.164	0.191	0.412
Safest	1.62E−4	0.459	0.018	0.077	0.409
Current Solution	9.80E−3	0.331	0.026	0.191	0.414
Selected Solution	1.62E−4	0.066	0.110	0.158	0.412

Table 5.6: *Quality attribute values for a selection of candidate solutions*

utilization), cost is shown in USD/1000, CPU utilization is shown as average value (1.0 represents maximum utilization), response time is shown as the average of response time relative to deadline for each task, and safety is shown as failure probability rate. When comparing the attribute values for the cheapest candidate, the safest candidate and the current solution in Table 5.6, it shows that the cheapest candidate is lowest in cost but worst in CPU utilization and response time, and that the safest candidate is best in safety but high in cost and a little worse than the current solution in CPU utilization.

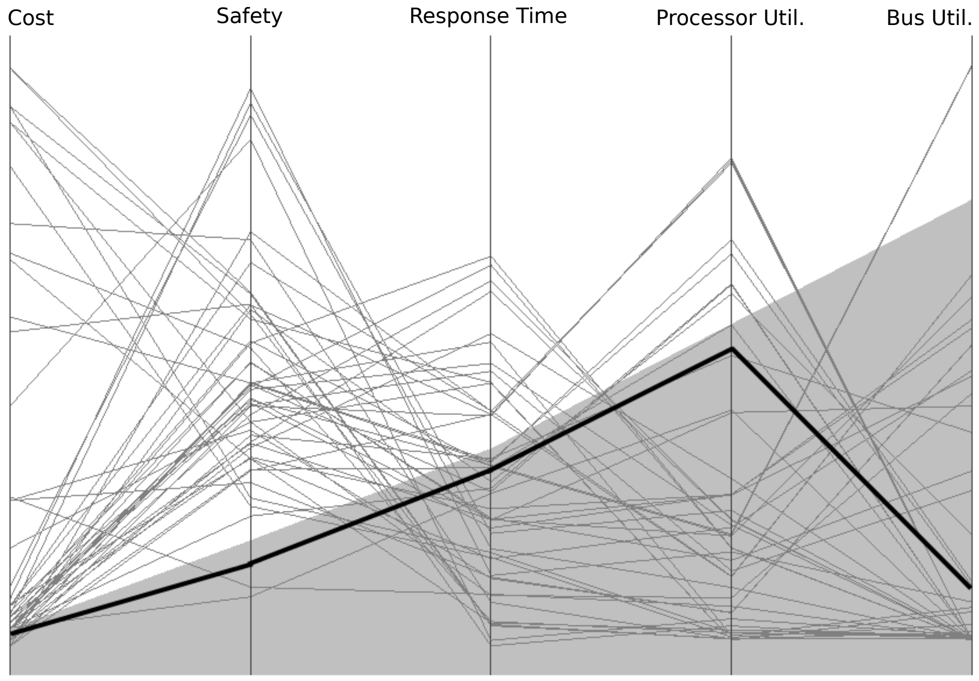


Figure 5.22: *Parallel coordinate plot of optimized solutions*

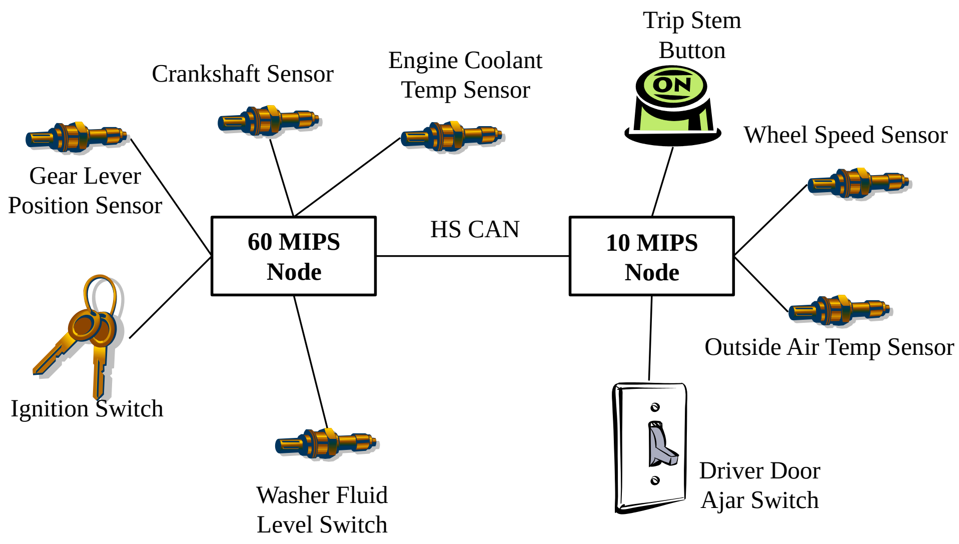


Figure 5.23: *Selected solution as proposed realization*

The goal of the case study is to find a solution that is cheaper than the current realization while fulfilling the requirements and constraints. The highlighted area in Figure 5.22 delineates the area that the architect considers suitable solutions based on quality attribute values, depicted in a parallel coordinate plot. After analysing the results of applying AQOSA and considering all attributes and constraints, an overall best solution was identified. In Figure 5.22, the bold line represents the selected solution. It contains 2 nodes: one 60 MIPS processor and one 10 MIPS processor, respectively. 12 components are deployed on the 60 MIPS node, and the remaining 6 components are deployed on the 10 MIPS node. The displays and gauges are part of the 60 MIPS node. The nodes are connected to a 500 kbps HS CAN bus. The selected solution is shown in Figure 5.23, and the attribute values are shown in Table 5.6. When comparing the attribute values of the current solution and the selected solution, it shows that the selected solution is significantly lower in cost, better in response time, and slightly better in safety. On the other hand, the selected solution is higher in CPU utilization than the current realization. This is not regarded as a problem, since the CPU utilization is still on a low level and the response time is low. The bus utilization is low in both solutions. So, it can be concluded that the selected solution is a cheaper and better realization, given the user functions and the attributes in the case study. This is also confirmed by the Pareto fronts in Figure 5.20 and Figure 5.21, which show that the selected solution (denoted by a black square symbol) is Pareto-optimal.

5.3.6 Validation and Discussion of Results

As mentioned earlier, the goal of this case study was to find a solution that is cheaper than the current realization while fulfilling the requirements and constraints. This is an ever-existing problem in the automotive industry, because of high cost pressure and variation in constraints during the lifetime of a system design. The product cost is important for every industrial sector producing embedded systems in large numbers, such as the automotive sector. The expected lifetime of an automotive system design is around 5 – 8 years. This lifetime is challenged by changes in constraints, such as optional functions becoming standard functions, or hardware parts becoming obsolete. Therefore, the problem studied in the case study is a real problem in the automotive industry.

We used the same requirement specifications for AQOSA as was used by the architects at Saab when designing the current realization. However, for practical reasons the focus of our case study research was on the most important user functions, and the most important constraints. Thus, the case study is based on requirement specifications for an actual industrial design problem.

The candidate architectures produced by the AQOSA optimization framework were presented to one of the main architects behind the current solution[2]. He agreed that the suggested solution by AQOSA is a suitable starting architecture for the next generation of cars at Saab.

A limiting factor for the accuracy of our results is the method to obtain the number of execution cycles for an operation. Components for which the source code was available, were analyzed with the SCoPE simulation framework [PHS+04], to compile the number of execution cycles. Components for which the source code was not available, were analyzed manually by reading the requirement specification and comparing this to similar components, to estimate the number of execution cycles. This was an efficient way of using the information that was available for each component. A quantitative accuracy assessment of the results from the AQOSA optimization framework could be obtained by measuring the quality attribute values of the real system and comparing to the simulated quality attribute values. This is part of the future work.

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

[2]The architect now works for a different company in the automotive sector.

5.3.7 Interpretation of Results

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

The case study shows the importance of considering all attributes and constraints when designing the architecture. If only a subset of the attributes are considered during design, there is a risk to select a solution that is infeasible with respect to other equally important attributes. Five quality attributes and three deployment constraints were considered during design. The theoretical design space is $64 \cdot 10^6$ given the repository of hardware components. Solving this design problem using the AQOSA framework required two sources of effor: a manual effort of 113 man hours in for creating the models of the components (this was largely done based on profiling of source code) in the repository as well as the behaviour models. Also, some man-hours were spent on analyzing the output of AQOSA. Additionally, 90 hours of continuous computer execution time was used for running the AQOSA optimization software. For a next design problem in the same domain the models of the components and behaviour can be reused with could reduce the time needed for making an analysis.

We found that the AQOSA framework supports the architect by proposing revolutionary [Axe09] [LH12] architectural solutions. A human architect normally uses previous architectures as a starting point for new architectures, which might prevent revolutionary ideas. However, the results from the case study show that AQOSA proposes very different solutions ranging from 1 hardware node up to 18 hardware nodes. Architects at Saab also confirmed that they considered AQOSA as a good tool for generating completely new solution ideas. Nevertheless, it should be noted that a human architect needs to assess and potentially modify the solutions proposed by AQOSA. Moreover, the framework uses plug-ins for analyzing several quality attributes simultaneously, which is important support because considering several quality attributes simultaneously is a difficult task for a human architect. Hence, by providing automatic and simultaneous analysis of quality attributes, AQOSA saves manual effort and development cost.

5.4 Summary

The studies in this chapter demonstrated the usefulness of software architecture optimization framework through 3 different case studies. These studies range from business information systems to embedded systems in the automotive industry. These case studies will be used as running case studies for next chapters as well.

The last case in this chapter reports on a real-world large-scale industrial case study applying a meta-heuristic optimization approach for automated software architecture design which supports multiple quality attributes. The case shows in an industrial context how meta-heuristic optimization approaches can improve software architecture design with respect to multiple quality attributes, and can suggest a wide range of optimized architectural solutions. Comparing the solutions proposed by AQOSA to the existing realization shows that AQOSA is able to synthesize efficient solutions in all quality attributes while fulfilling given constraints. Also, in contrast to human architects who tend to propose solutions based on previous architectures, AQOSA proposes revolutionary solutions.

Although the case study shows the saving of manual effort which is beneficial for time-to-market and development cost, it also shows that the proposed architecture solutions needs to be assessed by human architects. So, this chapter demonstrates how an architecture design framework like AQOSA complements the domain knowledge and experience of the architect.

To summarize, the case studies in this chapter show that meta-heuristic optimization can improve the process of designing efficient architectures for a set of given quality attributes in the following aspects:

- efficient search of large solution space
- considers multiple system quality attributes in design and analysis
- proposes revolutionary solutions

Chapter 6

New Degrees of Freedom for Software Architecture Optimization

This chapter addresses **RQ2** which is mentioned in Section 1.2:

> Can enlargement of the optimization search space help the meta-heuristic approach to find better architectural solutions?

We know the component-based paradigm makes it possible to easily and automatically create variation of architectural designs. Hence, the component-based paradigm is key to meta-heuristic optimization approaches, such as genetic algorithms (GA), to automatically generate new alternative solutions. However, to guarantee that the variation process does not change the functionality of the system, these approaches should only consider variations of architectural designs that do not modify the interfaces of components. The ways in which a candidate solution architecture can be varied without changing the functionality are called *Degrees of Freedom*.

This chapter presents two novel degrees of freedom for the optimization of software architectures. These two degrees of freedom are: (i) the topology of the hardware platform, and (ii) the replication of software components. The results show us that they can improve the results of the optimization algorithm by enlarging design space. This chapter analyses these two new degrees of freedom by running a very computationally-intensive experiment of an industrial case study.

This chapter is structured as follows. First, details of two new proposed degrees of freedom (DoFs) are discussed in Sections 6.1 and 6.2. After that, with two experiments based on two different systems, the effects of the proposed degrees of freedom for finding better solutions and enlarging design space have been analysed. These experiments and theirs results are presented in Sections 6.3, 6.4. Section 6.3 shows an experiment based on Cruise Control case study (see Section 5.2) which has been run with only topology DoF. Section 6.4 demonstrates a larger experiment which is based on SAAB Instrument Cluster case study (see Section 5.3) and has been conducted with both of the new proposed DoFs. Lastly, Section 6.5 summarizes this chapter.

6.1 New Degree of Freedom 1: Topology

We start with an example in order to provide a better describe for the topology-DoF: The case study of [MKBR10] demonstrates the state-of-the-art optimization based on an initial architecture. Figure 6.1 represents the base topology used in their study. In their approach, new alternative solutions can only be created by leaving out nodes and/or buses from this base topology. Hence the manner in which nodes can be connected is 'hard-wired' into the initial base topology. Hence in this setting, the optimizer can achieve architectures with topologies shown in Figures 6.2a, 6.2c and 6.2e. For example, to achieve topology that depicted in Figure 6.2a the optimizer should keep *Node1*, *Node2*, *Node3*, *Node5*, *Node7* and *Bus1*, *Bus2*, *Bus3* and should remove *Node4*, *Node6*, *Node8* and *Bus4*. Similar to this, to achieve Figure 6.2c the optimizer keeps *Node2*, *Node3*, *Node5*, *Node7*, *Node8* and *Bus2*, *Bus3*, *Bus4* and removes *Node1*, *Node4*, *Node6* and *Bus1*.

The topologies shown in figures 6.2b, 6.2d and 6.2f can not be created from the base topology. For example, in Figure 6.2d if *Bus2* is used, it must be connected to all of predefined nodes – as is the case in the base topology. In the existing optimization approaches a bus is either in or out of the architecture. Using our new DoF, we can create new topologies that contain new nodes and new connections that were not yet in the original base topology. In theory this makes the search space infinitely large. However, we use use pragmatic bounds on the size of the solution. These bounds still allow more variation of solutions that the state-of-the-art, yet limit number of computations needed by the optimizer within practical scope.

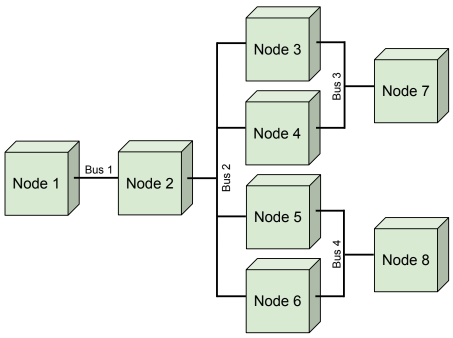
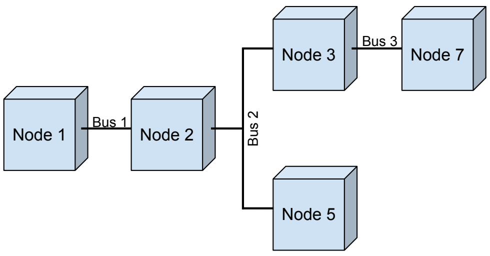
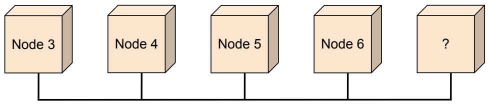


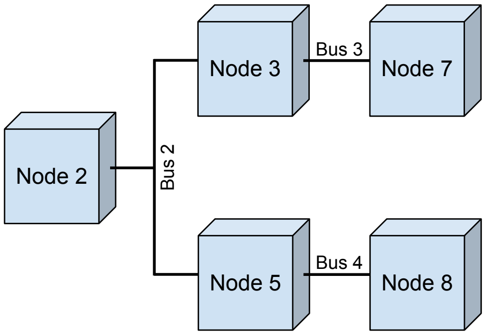
Figure 6.1: *Base topology*



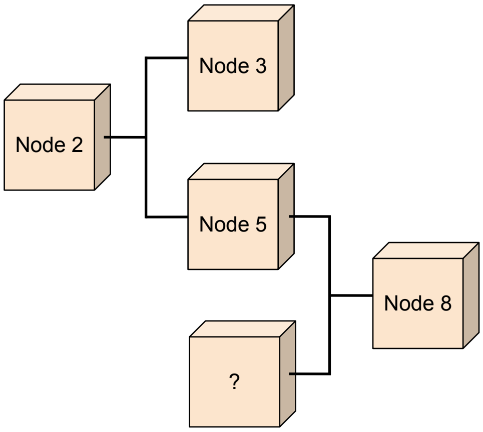
(a) *Possible topology 1*



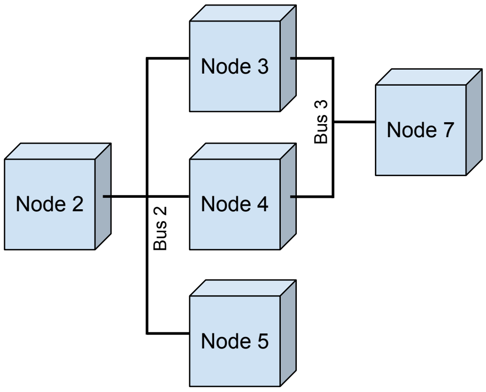
(b) *Impossible topology 1*



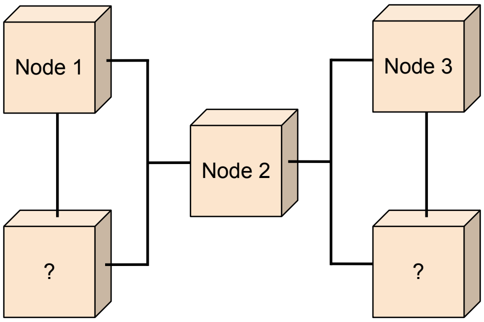
(c) *Possible topology 2*



(d) *Impossible topology 2*



(e) *Possible topology 3*

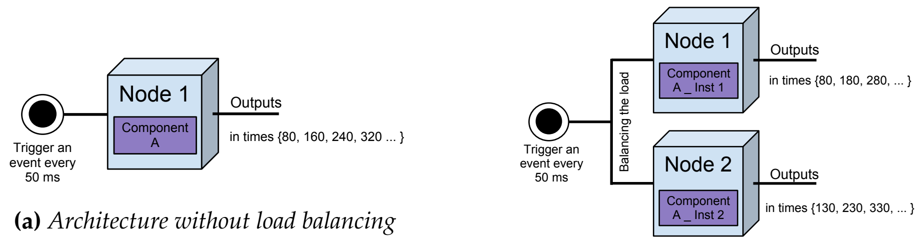


(f) *Impossible topology 3*

Figure 6.2: *Samples of possible and impossible topologies*

6.2 New Degree of Freedom 2: Replication of software components

We use an example in order to describe the replication-DoF: Assume an event in a simple system is being called every 50ms and its deadline is 100ms. The execution time of the related component on the most powerful processor from repository is 80ms. Because the execution time is larger than the trigger time, the requests miss their deadlines. In this kind of situations, the replication-DoF opens the possibility to solutions where the processing is distributed across multiple processor nodes running in parallel. This allows for a common architectural solution called 'load balancing'. By applying the load balancing technique, the system can handle multiple request in parallel and thus increase the throughput of the system.



Node 1
Component A
Outputs
in times {80, 160, 240, 320 ... }
Trigger an event every 50 ms
Balancing the load
Node 1
Component A _ Inst 1
Outputs
in times {80, 180, 280, ... }
Node 2
Component A _ Inst 2
Outputs
in times {130, 230, 330, ... }

(a) *Architecture without load balancing*

(b) *Architecture with load balancing DoF*

Figure 6.3: *Comparing the effect of load balancing DoF*

These quality properties could be critical design objectives in business- or safety-critical systems. Because without having such degree of freedom in generating new solutions and its related mechanism in performance evaluation, the optimizer generates solutions which possibly miss some deadlines depends on defined scenarios and load of the system. For example, in the aforementioned system, if the architecture contains two instances of one software component, then it can respond to all requests in a way which meets all deadlines. Figure 6.3 shows the topology that corresponds with the load balancing pattern in this example.

Because the load balancing technique distributes the load of the system, it affects the architecture evaluation scores for performance, reliability and utilization. However, it should be noted that the replication pattern is different from the Voting mechanism as described in e.g. [LSBB03]. Hence, it can not improve the safety aspect. When a node in the architecture fails, the system will miss all the assigned requests to that specific node. So, the other duplicated components do not cover for that failed node.

6.3 Experiment 1: Cruise Control System

The cruise control (CC) system is a safety critical system from the automotive industry. Details of this system are described in Section 5.2. Our version of it has been derived from the VehicleControl system from [GH11], which is based on a system in [HF07]. The main purpose of the cruise control is maintaining a predetermined constant speed which is set by the driver. The software architecture of the CC system consists of six software components: `IN_CONTROL`, `DESIRE_SPEED`, `AIRBAG`, `SECURITY BELT`, `THROTTLE`, and `BRAKE`.

The goal of this experiment is to examine the usefulness of our new topology degree of freedom. Hence, the experiment was defined in a way to generate a comparison between optimization with and without the topology DoF. Moreover, since this system is rather small, it is easy to depict the achieved optimal architectural solutions and thereby see its coverage of various topologies in the design space.

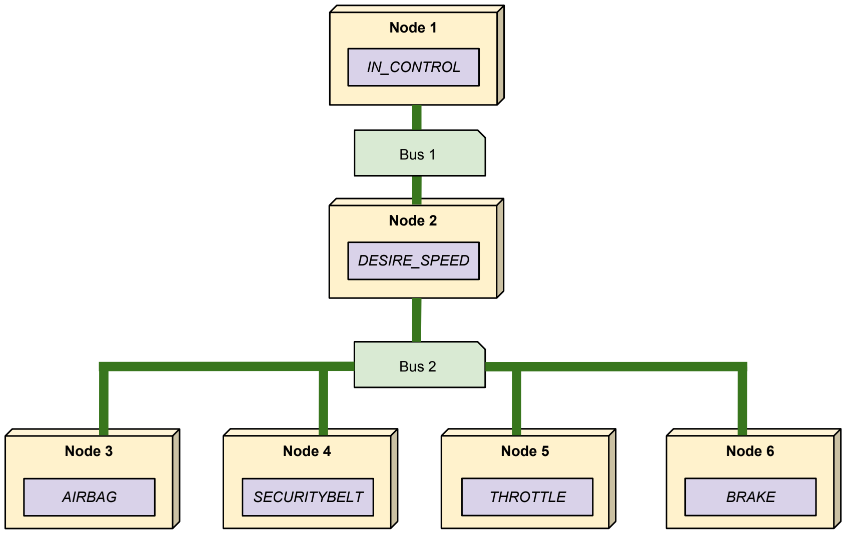


Figure 6.4: *Baseline topology for CC system*

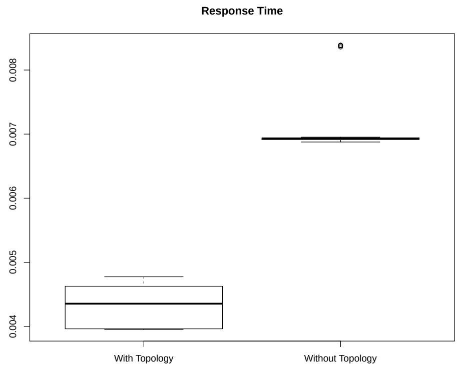
6.3.1 Optimization Setup

For solving the CC system design problem, a repository which consists of the following hardware components, has been considered:

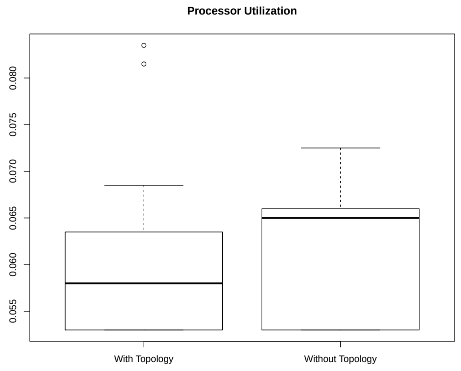
- 28 Processors types: ranging over 14 various processing speeds from 66MHz to 500MHz. Each of these has two levels of energy consumption. A processor is more expensive if it has a smaller chance of failure.

- 15 Buses: ranging over five types of bandwidths (128, 160, 192, 224 and 256 kbps) and each type has three variants with different latencies (1, 3 and 5 ms). Again, a bus is more expensive if it supports higher bandwidth or smaller time delay.

The AQOSA optimization was run with the following parameter settings: initial population size (α) = 100, parent population size (μ) = 50, number of offspring (λ) = 50, archive size (with crowding type) = 50, number of generations = 10000, crossover rate is set to 0.95. We ran this experiment 50 times in total (25 times for both with- and with-out topology DoF). In the optimization without the topology DoF we used the baseline topology depicted in Figure 6.4. On the other hand, in the optimization with the topology DoF, the approach generates new topologies by itself and no baseline was used. This alleviates the architects from creating a based line architecture.



(a) *Best Response Time*



(b) *Best CPU Utilization*

Figure 6.5: *Boxplot comparison of optimization results with and without topology DoF*

6.3.2 Experiment Results

Figure 6.5 shows the difference between optimization with and without the topology DoF. It shows the boxplot chart of 25 runs with topology DoF and 25 runs without it, for response time and processor utilization objectives. In both graphs, lower values for objectives indicate better solutions. Figure 6.5a demonstrates that the with-topology approach converges to solutions that are approximately 35% better for response time. Also Figure 6.5b demonstrates that optimizing with topology DoF finds solutions with on average a 12% better system processor utilization.

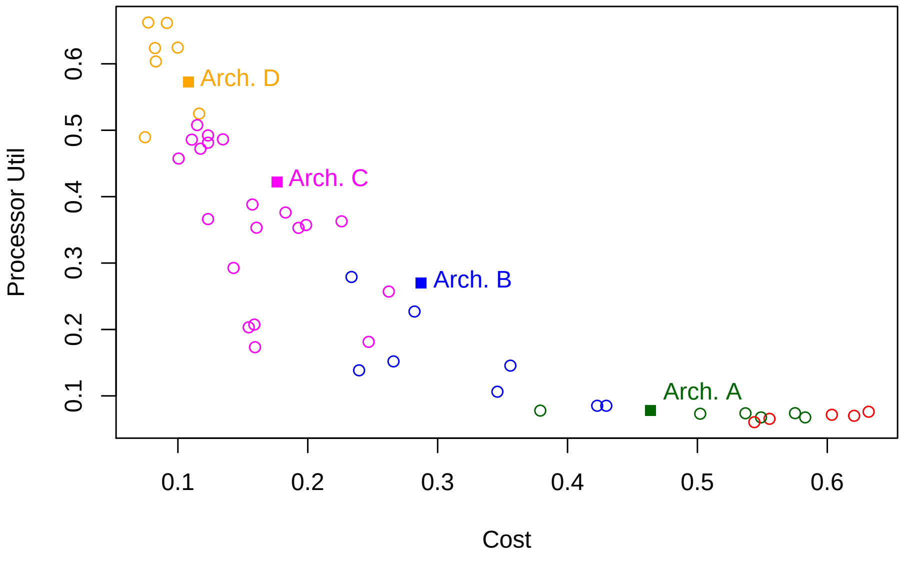


Figure 6.6: *Pareto front of Processor utilization vs. Cost*

Because the optimization of this case study considers five objectives, there are 10 2D Pareto fronts for different pairs of objectives. Figure 6.6 shows one of these fronts for the objectives *Processor utilization* and *Cost*. Each point in the plot depicts an architectural solution. The 2D Pareto fronts show the trade-off between objectives. Improving one dimension often implies a decrease of the other dimension. Different colors in the plots correspond to different numbers of nodes in the architecture: black for 1-node, orange for 2-node, magenta for 3-node, blue for 4-node, green for 5-node and red for 6-node architectures. As can be seen there is no black node which means AQOSA could not find an optimal solution with a 1-node architecture.

Moreover, to demonstrate that the approach can generate various types of topologies, one 5-node architecture (Figure 6.7), one 4-node architecture (Figure 6.8), one 3-node architecture (Figure 6.9) and one 2-node architecture (Figure 6.10) out of 50 optimized solutions have been chosen. These are highlighted in Figure 6.6 with the filled squares as well.

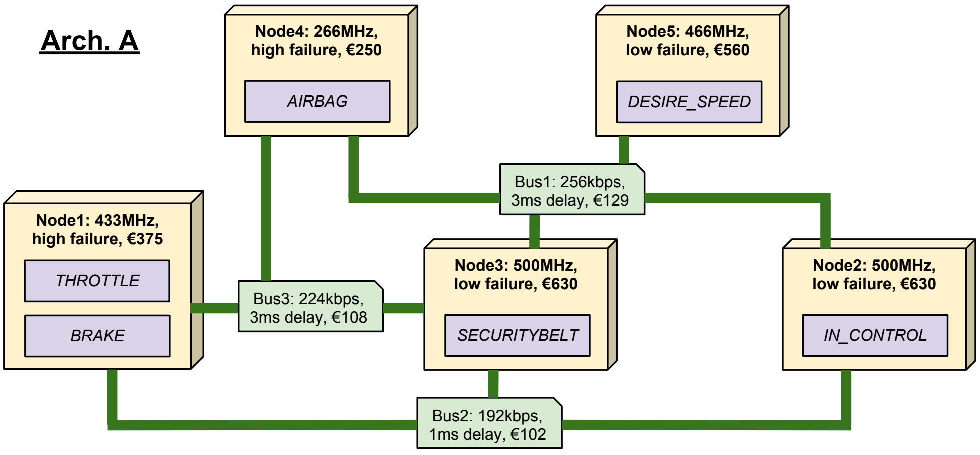

Arch. A
Node4: 266MHz, high failure, €250
AIRBAG
Node5: 466MHz, low failure, €560
DESIRE_SPEED
Bus1: 256kbps, 3ms delay, €129
Node1: 433MHz, high failure, €375
THROTTLE
BRAKE
Bus3: 224kbps, 3ms delay, €108
Node3: 500MHz, low failure, €630
SECURITYBELT
Node2: 500MHz, low failure, €630
IN_CONTROL
Bus2: 192kbps, 1ms delay, €102

Figure 6.7: *Arch. A: Sample of 5-node topology*

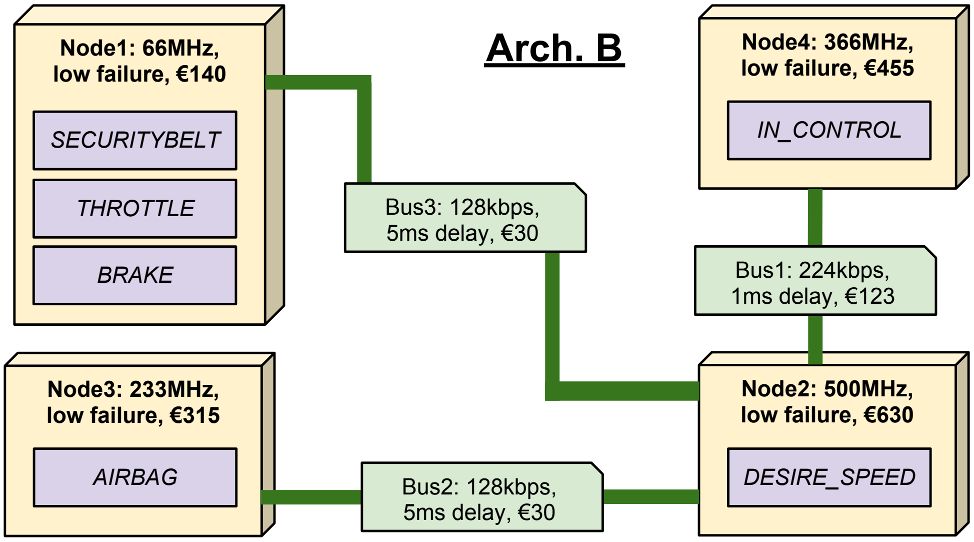

Node1: 66MHz, low failure, €140
SECURITYBELT
THROTTLE
BRAKE
Arch. B
Node4: 366MHz, low failure, €455
IN_CONTROL
Bus3: 128kbps, 5ms delay, €30
Bus1: 224kbps, 1ms delay, €123
Node3: 233MHz, low failure, €315
AIRBAG
Node2: 500MHz, low failure, €630
DESIRE_SPEED
Bus2: 128kbps, 5ms delay, €30

Figure 6.8: *Arch. B: Sample of 4-node topology*

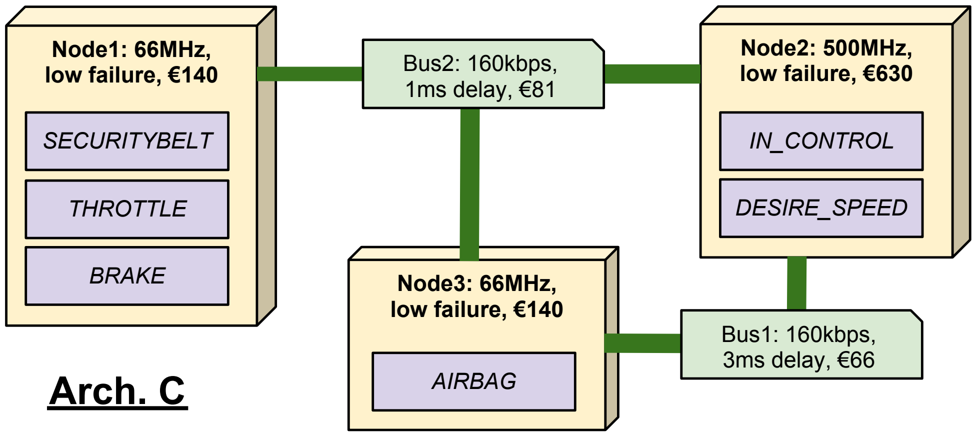

Node1: 66MHz, low failure, €140
SECURITYBELT
THROTTLE
BRAKE
Bus2: 160kbps, 1ms delay, €81
Node2: 500MHz, low failure, €630
IN_CONTROL
DESIRE_SPEED
Node3: 66MHz, low failure, €140
AIRBAG
Bus1: 160kbps, 3ms delay, €66
Arch. C

Figure 6.9: *Arch. C: Sample of 3-node topology*

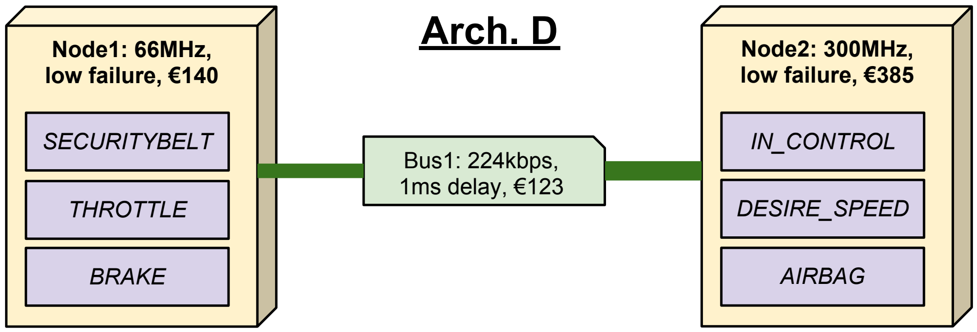

Arch. D
Node1: 66MHz, low failure, €140
SECURITYBELT
THROTTLE
BRAKE
Bus1: 224kbps, 1ms delay, €123
Node2: 300MHz, low failure, €385
IN_CONTROL
DESIRE_SPEED
AIRBAG

Figure 6.10: *Arch. D: Sample of 2-node topology*

6.4 Experiment 2: The SAAB Instrument Cluster System

This experiment is conducted based on the real world industrial system of the SAAB Instrument Cluster sub-system. The system is described in Section 5.3. It consists of 18 components as depicted in Figure 5.8. The purpose of the Instrument Cluster sub-system is to provide the driver with information that is required when driving the car. The information is processed by user functions based on sensor values and presented on gauges or displays located in front of the driver.

The goal of this computationally-intensive experiment is to examine the usefulness of both of the two new proposed degrees of freedom within the software architecture optimization process. Therefore, for this reason two comparison were targeted for this experiment: (i) comparing of best of achieved values for different objective functions, (ii) comparing the hypervolume indicators of the entire sets of optimal solutions.

6.4.1 Optimization Setup

For solving the Instrument Cluster system design problem, a repository that can use the following hardware components has been considered:

- 22 types of Processor nodes: ranging over 14 various processing speeds from 66MHz to 400MHz. Each of these has two levels of probability of failure. A processor is more expensive if it has a smaller chance of failure. (See lines 368 to 389 in Listing A.1)
- 4 types of Bus nodes: ranging over bandwidth (10, 33, 125, and 500 kbps) and each type has different latency. Again, a bus is more expensive if it supports higher bandwidth and less time delay. (See lines 390 to 393 in Listing A.1)

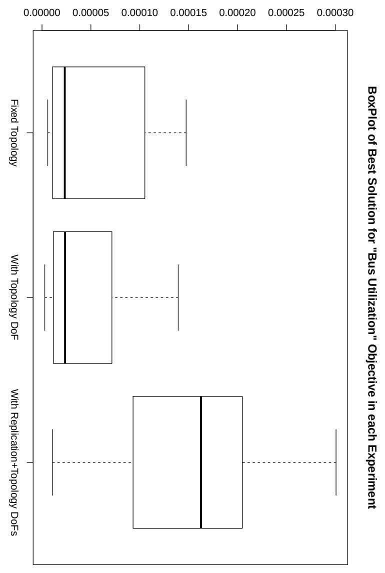
After defining the above hardware options, AQOSA was run with the following parameter settings: initial population size (α) = 1000, parent population size (μ) = 100, number of offspring (λ) = 50, archive size = 50, number of generations = 5000, crossover rate = 0.95, constant mutation probability = 0.01.

For comparing optimization with and without these new DoFs, three separate sets of experiments set have been defined:

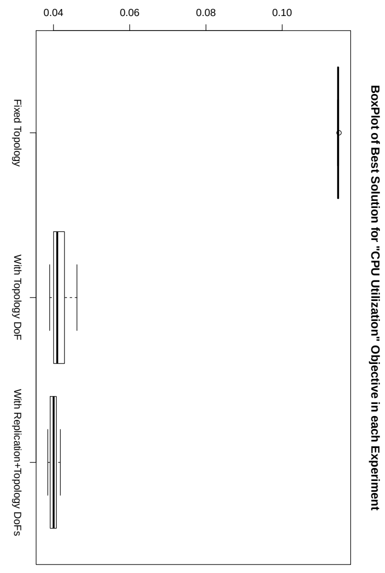
1. **Experiment Set (1)**:
   Optimization without topology- and replication- DoFs. AQOSA supports general DoFs which are mentioned in Section 4.4.2, except topology DoF.
2. **Experiment Set (2)**:
   Optimization with topology degree of freedom. AQOSA support all DoFs which are mentioned in Section 4.4.2 but not replication-DoF.
3. **Experiment Set (3)**:
   Optimization with topology- and replication- degrees of freedom. AQOSA support all DoFs which are mentioned in Section 4.4.2 plus replication-DoF.

Each experiment set was run 30 times (90 experiments in total). For running experiment set (1), the baseline topology depicted in Figure 5.19 was used. On the other hand, for experiments sets (2) and (3), the AQOSA framework generates topologies by itself. In other words, they are not based on any baseline architecture.

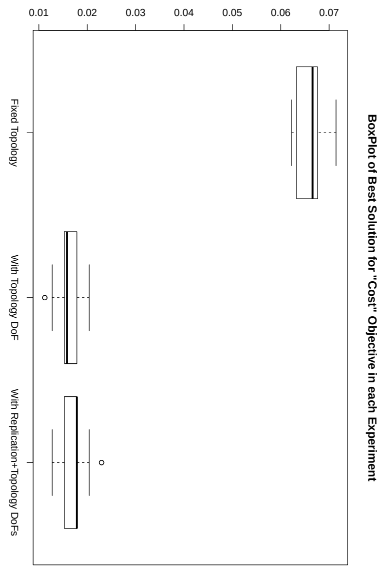
Running on a powerful computer with a 12-core processor (each core runs 2.67GHz and 12MB cache) and 48GB memory, each experiment took between 1 to 2 hours. In total, the whole experiment took nearly 1 week of continuous computation.



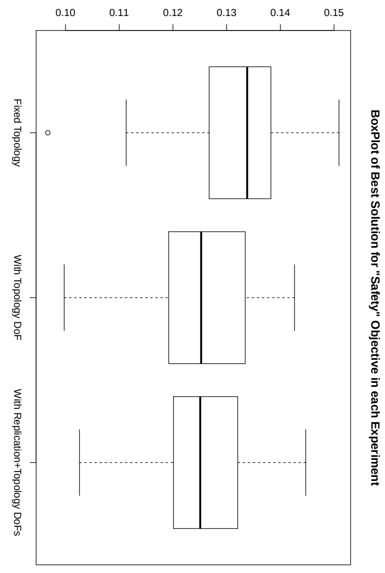
(a) *Best of Bus utilization*



(b) *Best of CPU utilization*



(c) *Best of Cost*



(d) *Best of Safety*

Figure 6.11: *Boxplot comparison of three experiments sets*

6.4.2 Experiment Results

Figure 6.11 shows the differences between optimization with and without new DoFs. It shows the boxplot chart of 30 runs for each experiment set. It demonstrates the achievement of best solution in different objectives. In all of four graphs, lower values for objectives indicate better solutions because the optimization process was set to minimize all objectives values.

From these graphs, different effects for different objectives can be observed. For example, it shows us that for the processor utilization and cost objectives, the two new degrees of freedom significantly (approximately 65% and 75%, respectively) improve the results found by the optimization process: there are quite big improvement in these two objectives. For the objective safety, the two new DoFs slightly help in finding better solutions. And finally, for the objective bus utilization no improvement can be discovered as the results of using the two new degrees of freedom.

We expected that if the replication-DoF would have an effect, then this effect would mostly likely be large for the CPU utilization objective. Hence, to analyse the effectiveness of this DoF we compare the CPU utilization of experiment set (2) with that of experiment set (3). However, in Figure 6.11b both are far better than the results of experiment set (1). Figure 6.12 shows the boxplot chart of best CPU utilization in 30

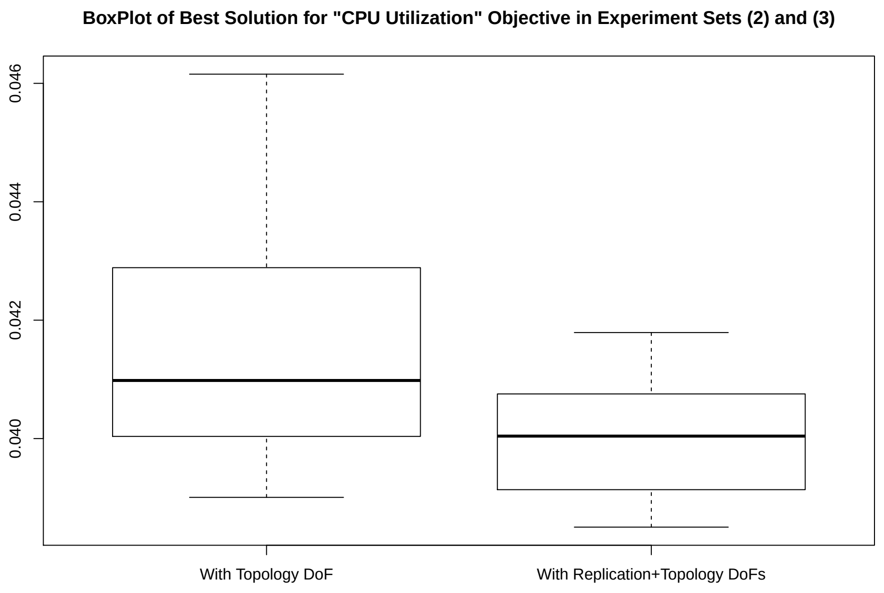


Figure 6.12: *Best of Processor utilization for experiment sets (2) and (3)*

runs only for experiment sets (2) and (3). Again, a lower value indicates a better CPU utilization. This means that running the algorithm with this DoF ended up with better solutions regarding the CPU utilization objective. Hence, the graph demonstrates notable improvement for the CPU utilization objective by using the replication-DoF.

To represent the coverage of all of five objectives, Figure 6.13 depicts a comparison of hypervolume indicators for three experiment sets. In this graph, higher values mean better coverage of the design space. We set reference points in hypervolume calculation to 2 (all values are between 0 and 1), because fix topology optimization concentrate on one region but optimization with new DoFs try to find solutions even in the boundary regions. So, reference points were defined a little far from boundaries. As can be seen in the graph, two experiment sets with the new DoFs perform considerably better than experiment set (1). Also, since the boxplot of experiment set (1) shows a smaller variance, it can be concluded that this approach finds solutions from a more limited area of the design space. Of course, experiment sets (2) and (3) depend, like any genetic algorithm, on sufficient randomness in the optimization process in order to achieve good coverage of the design space and hence in finding optimal solutions.

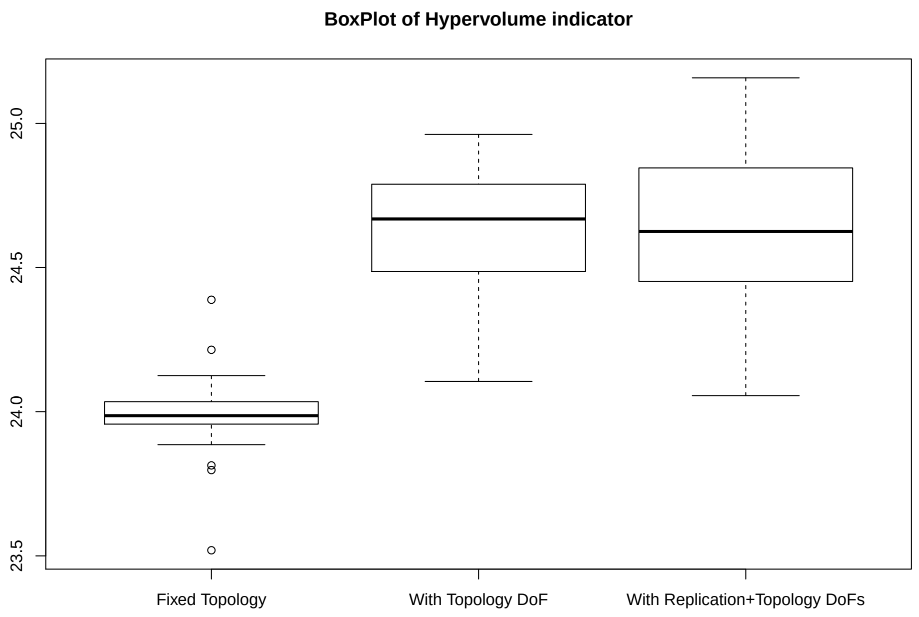


Figure 6.13: *Boxplot comparison of the hypervolume of archive sets*

6.5 Summary

This chapter studied the improvements that can be obtained by introducing new degrees of freedom in the architecture representation (**RQ2**). In this chapter two novel degrees of freedom were introduced: (1) a new method for varying the topology of system architectures, (2) allowing architecture to replicate software component instances. More degrees of freedom essentially enlarge the search space, and hence allow evolutionary algorithms to find better solutions. By running a very computationally-intensive experiment on an industrial case study, it was shown that optimization using this approach indeed finds better system architectures. These experiments bring empirical evidences that prove better solutions can be found by using these new degrees of freedom. Moreover the approach was still computationally feasible.

This method also opens the doors for the next research question in the next chapter: We would like to introduce knowledge-directed search. To this end, we envision that we first diagnosing bottlenecks in an architecture design and then apply suitable architecture tactics as transformation of this bottleneck of the architecture which removes or reduces this bottleneck. Topology variation is fundamental for this method because it is not possible to apply many common tactics without having the topology DoF in the software architecture. Moreover, using the DoF introduced in this chapter it is possible to develop a load balancing search operator and apply it in a smarter targeted manner. This approach could be very useful in design problems with hard deadline such as embedded systems.

Chapter 7

Heuristic-based Application of Search Operators

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

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

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

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

7.1 Problem-Specific Search Operators

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

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

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

7.1.1 Caching for improving performance

Basically, there are two main caching strategies that can be described as patterns [Rot06]: Primed Cache and Demand Cache. If the data required for performing a certain computation is known prior to the start of that computation, then the system

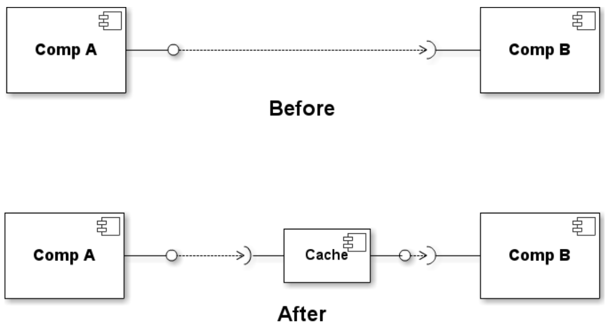


Figure 7.1: *Caching pattern*

can store it before the computation starts, which is called Primed Cache. However, in case the data required by a computation can vary for each run, the system can bring the data into the memory whenever required and keep it for future use. This strategy is called the Demand Cache pattern.

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

7.1.2 Voter pattern for improving safety

Masking faults is one of the primary approaches to improve the behaviour of a system in a faulty environment. N-modular redundancy and N-version programming are

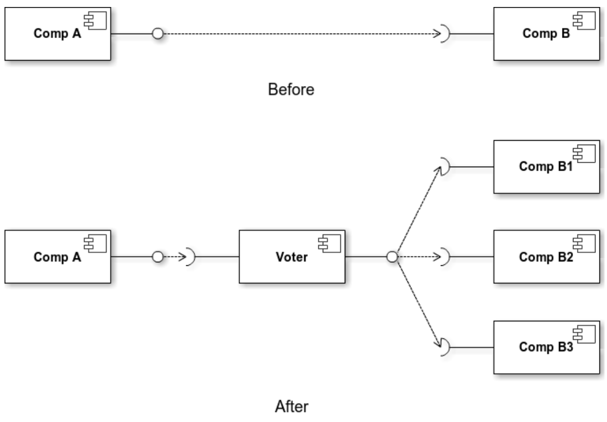


Figure 7.2: *Voter pattern*

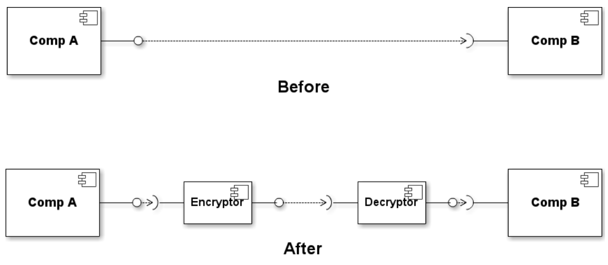


Figure 7.3: *Encryption/Decryption pattern*

well-known fault masking methods. These approaches use redundant modules and a voting unit to hide the occurrence of errors. The voter arbitrates between the achieved results and produces a single output [LSBB03].

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

7.1.3 Encryption/Decryption for improving security

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

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

7.2 Heuristic-based Search Operators by Anti-patterns

Architecture- and design-patterns capture expert knowledge about "best practices" in software design by documenting general solutions that may be customized for a particular context. They make it possible to reuse the knowledge of software design and to focus on quality attributes such as performance. Software anti-patterns are conceptually similar to patterns in that they document recurring solutions to common design problems (i.e. the "bad practices") as well as their solutions: what to avoid and how to solve the problems [CMT10].

Software architecture design patterns look at the positive and constructive features of a software system, and suggest common solutions. In contrast, anti-patterns look at the negative and destructive features of a software system, and present common solutions to the problems that make negative consequences. Bottlenecks affect quality attributes negatively. Therefore we explore an approach that first detects anti-patterns and used these as indicators of possible bottlenecks in architectural solutions. Next, a suitable transformation needs to be applied to remove/reduce the suspected bottleneck. In our study we consider four architecture heuristics as problem-specific search operators. The first two operators are derived from the *Concurrent Processing Systems* anti-pattern. As it is stated in [Tru11], *"[This anti-pattern] occurs when processing cannot make use of available processors"*. In other words, the processes running on the system cannot use the available resources effectively. This could happen when the processes are assigned to the processors in a non-balanced way. The other operators address other quality attributes: cost and reliability. We explain these in more detail in the subsequent sections.

7.2.1 Search Operator: Component Movement

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

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

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

7.2.2 Processor Change for Performance

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

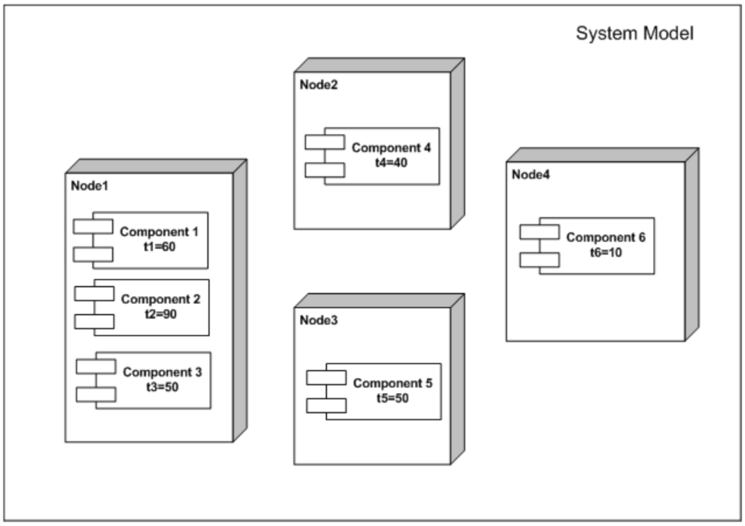


Figure 7.4: *Non-balanced distribution of software components*

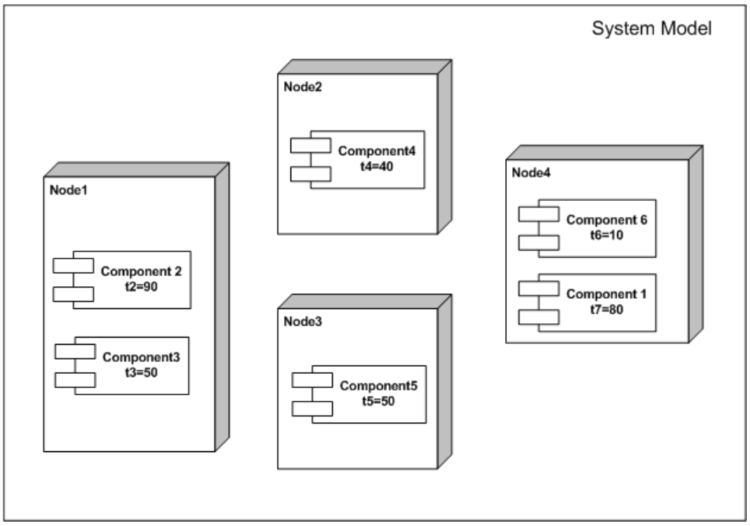


Figure 7.5: *Balanced distribution of software components*

7.2.3 Processor Change for Cost

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

7.2.4 Processor Change for Reliability

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

7.3 Combining Search Operators in the Genetic Algorithm

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

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

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

$$\text{Operators' set} = \{'swMove', '\mathtt{pc4Perf}', '\mathtt{pc4Cost}', '\mathtt{pc4Rely}'\} \tag{7.1}$$

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

$$\mathtt{GE}() : \text{return generic GA operator.} \tag{7.2}$$

$$\mathtt{RA}(set) : \text{return random operator out of the set.} \tag{7.3}$$

$$\mathtt{RR}(set) : \text{return an operator in round-robin order out of the set.} \tag{7.4}$$

Finally, for generating offspring we need to call two operators which are chosen by one of aforementioned functions:

$$\begin{gathered}\mathsf{Offspring}(\mathsf{Parents}) = \{\mathsf{Child}_1, \mathsf{Child}_2\}, \text{ where:} \\ \mathsf{Child}_1 = \mathsf{Operator1}(\mathsf{Parent}),\ \ \mathsf{Child}_2 = \mathsf{Operator2}(\mathsf{Parent}).\end{gathered} \tag{7.5}$$

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

7.3.1 Random

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

$$\mathsf{Operator1} = \mathsf{RA}(\mathsf{set}),\ \ \mathsf{Operator2} = \mathsf{RA}(\mathsf{set}). \tag{7.6}$$

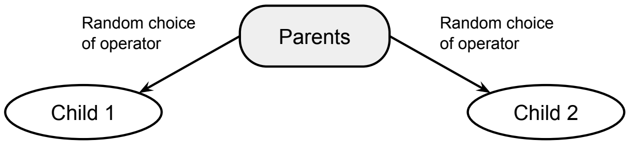


Figure 7.6: *Random combination of heuristic-based search operators*

7.3.2 Sequential

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

$$\mathsf{Operator1} = \mathsf{RR}(\mathsf{set}),\ \ \mathsf{Operator2} = \mathsf{RR}(\mathsf{set}). \tag{7.7}$$

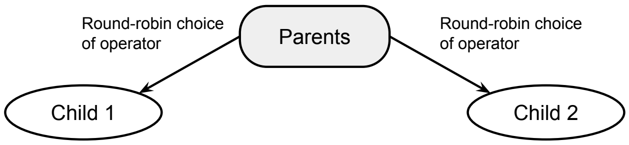

Round-robin choice of operator
Parents
Round-robin choice of operator
Child 1
Child 2

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

7.3.3 Random-Sequential

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

$$\text{Operator1} = \text{RA}(\text{set}), \;\; \text{Operator2} = \text{RR}(\text{set}). \tag{7.8}$$

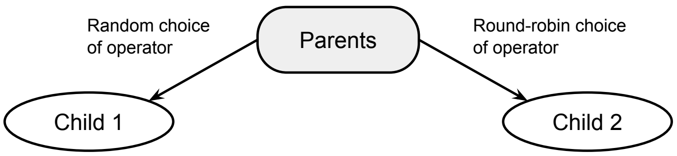


Figure 7.8: *Random-Sequential combination of heuristic-based search operators*

7.3.4 Half-Random

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

$$\text{Operator1} = \text{GE}(), \;\; \text{Operator2} = \text{RA}(\text{set}). \tag{7.9}$$

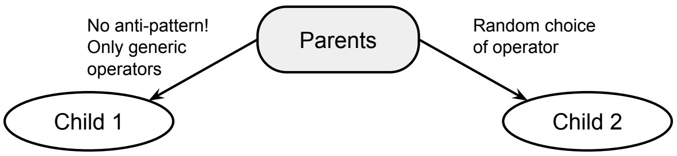

No anti-pattern! Only generic operators
Parents
Random choice of operator
Child 1
Child 2

Figure 7.9: *Half-Random combination of heuristic-based search operators*

7.3.5 Half-Sequential

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

$$\text{Operator1} = \text{GE}(), \;\; \text{Operator2} = \text{RR}(\text{set}). \tag{7.10}$$

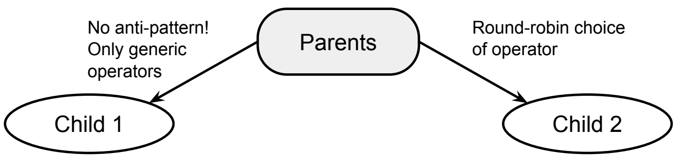


Figure 7.10: *Half-Sequential combination of heuristic-based search operators*

7.3.6 Half-Random-Sequential

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

$$\mathrm{Operator1} = \mathrm{GE}(), \ \ \mathrm{Operator2} = \begin{cases} \mathrm{RA}(set) & \text{if generation is even} \\ \mathrm{RR}(set) & \text{if generation is odd} \end{cases} \tag{7.11}$$

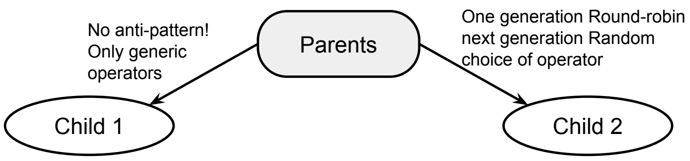


Figure 7.11: *Half-Random-Sequential combination of heuristic-based search operators*

7.4 Experiment

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

The goal of the experiment is to compare the combinations of operators, in terms of the speed by which they find optimal solutions. To this end, we defined an experiment with the following steps:

1. First we try to find out what would be the ideal Pareto front. Given that this may take very long to computer, we use the following as an approximation of this: We run the optimization process with only the generic search operators for a very high number of generations. This gives the algorithm enough opportunity to approximate the ideal Pareto front within a small margin. We used this set of solutions as the reference Pareto front for comparison with other strategies for combining search operators.

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

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

7.4.1 Experiment Setup

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

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

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

$$\max\left(\left(\frac{1}{N}\sum_{i=1}^{N}\mathrm{dist}(x_i, Y)^p\right)^{1/p}, \left(\frac{1}{M}\sum_{i=1}^{M}\mathrm{dist}(y_i, X)^p\right)^{1/p}\right) \tag{7.12}$$

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

For generating architectural solutions, the set of software components is given by the SAAB instrumentation cluster software architecture. Note that these components may be replicated in any actual architecture solution. In addition, the repository contains the following hardware components that may be used:

- 28 Processors: ranging over 14 various processing speeds from 66MHz to 500MHz; Each of them has two levels of failure rate. A processor is more expensive if it has less chance of failure and vice versa.
- 4 Buses: with bandwidths of 10, 33, 125, and 500 kbps, and latencies of 50, 16, 8, and 2 ms. A bus is more expensive if it supports higher bandwidth.

7.4.2 Experiment Results

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

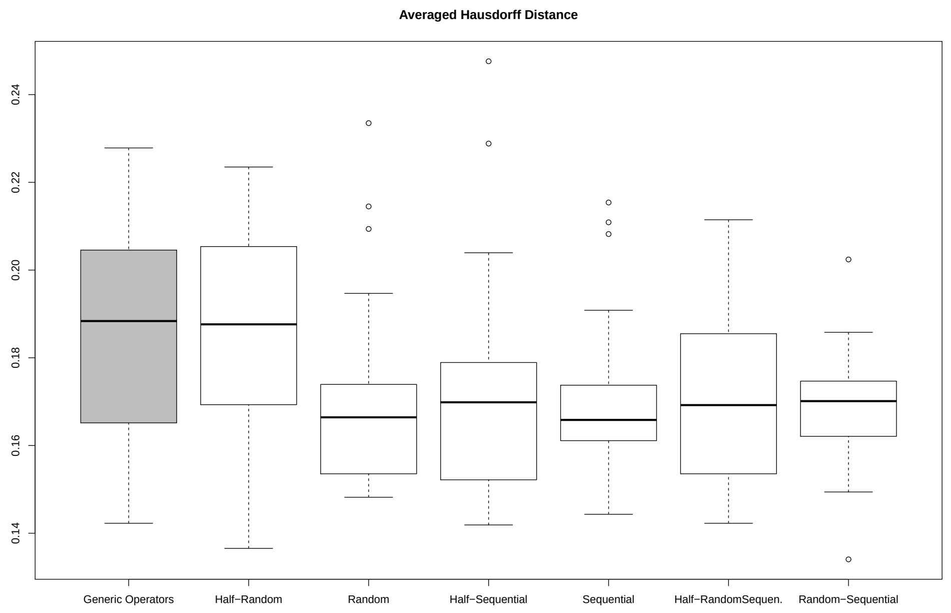


Figure 7.12: *Averaged Hausdorff Distance of different operator combinations*

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

7.5 Summary

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

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

Chapter 8

Software Architecture Optimization for Software Product Lines

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

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

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

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

8.1 Optimization for SPL

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

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

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

8.1.1 Process

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

1. Feature modeling
2. Architecture modeling
3. Connecting features to their implementing software components
4. (Optionally) Defining target products in the product line
5. Evolutionary optimization of software architecture for every product
6. Commonality Analysis among the Pareto front solutions of all products

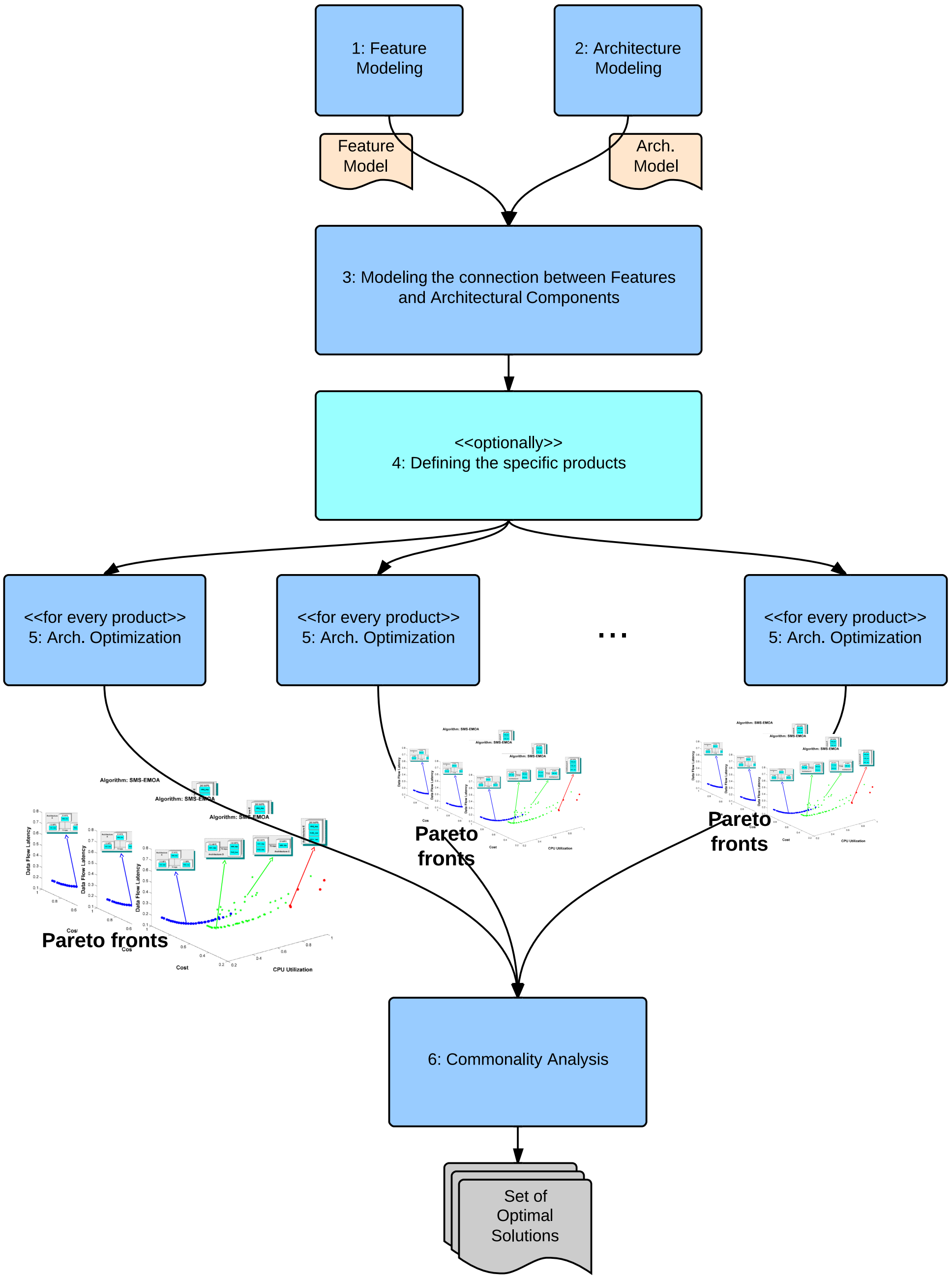


Figure 8.1: *Process of finding similar optimal architectural solutions from Pareto fronts*

Step 1 and *Step 2* are for designing a feature model and an architecture model, respectively. In *Step 3*, we define the relationship between each feature in the feature model and the software components in the architecture model, to specify which components provide the required functionality of that feature.

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

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

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

Listing 8.1: *Clafer model*

```
car
  xor ignition_switch
      key_ignition
      button_ignition
  interior_lights
  xor dashboard
      simple_dashboard
      extended_dashboard
  airbag_system ?
    front_airbag
    sides_airbag ?
    passengers_airbag ?
  antilock_braking_system ?
  traction_control_system ?
  xor stability_control_system ?
      basic_skid_control
      extended_skid_control
  xor cruise_control ?
      basic_cruise_control
      adaptive_cruise_control
      fullyAdaptive_cruise_control
  theft_alarm ?
  park_assist ?
[stability_control_system => traction_control_system]
[traction_control_system => antilock_braking_system]
[basic_cruise_control => basic_skid_control]
[adaptive_cruise_control => extended_skid_control]
[fullyAdaptive_cruise_control => extended_skid_control]
[button_ignition => !simple_dashboard]
```

8.1.2 Modelling

Feature modelling

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

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

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

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

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

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

Architecture modelling

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

Connecting features to their underlying software components

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

Defining target products in the product line

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

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

8.1.3 Optimization

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

8.2 Commonality Analysis

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

```
Input: all Pareto fronts (frontSets)
Output: a list of architectural solutions
foreach Solution s1 in one Front from frontSets do
    foreach Front f in remaining Fronts do
        foreach Solution s2 in f do
            calculate the distance between s1 and s2 ;
            if distance < $\Delta$ then
                break and continue with next Front f;
            end
        end
        break and continue with next Solution s1;
    end
    Store s1 in the list of common solutions;
end
```

Algorithm 1: Commonality analysis algorithm

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

8.2.1 Solutions Distance Calculation

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

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

```
Input: Two Solutions ($s1$ and $s2$)
Output: an integer number as their distance
store two solutions as list of resources;
find which list is longer ($longer$ and $shorter$);
foreach Resource $r$ in $longer$ list do
    if $shorter$ list contains $r$ then
        remove $r$ from both list;
    end
end
substitution changes = the remaining number of resources in $shorter$ list;
addition changes = (the remaining number of resources in $longer$ list) -
substitution changes;
distance = (substitution changes) + ($\omega$ * addition changes);
```

Algorithm 2: Solutions distance calculation algorithm

tions. Therefore, the algorithm also considers weights for the changes. So, we set the cost of substitution change such that it is always lower than the cost of an addition and/or removal change. In particular, we defined ω as the weight for this ratio. It is required to configure ω as a parameter of the algorithm. Algorithm 2 details pseudo codes of this algorithm.

8.3 Experiment for SPL-aware Optimization

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

Figure 8.2 depicts the feature model of the case study (Listing 8.1 shows its textual representation). A car in our model should consist of at least an ignition switch, interior lights, and dashboard. In addition, it may optionally have the following systems: airbag system, anti-lock braking system, traction control system, stability control system, cruise control system, theft alarm system and park assist system. The ignition switch can be either key ignition or button ignition technology. Likewise,

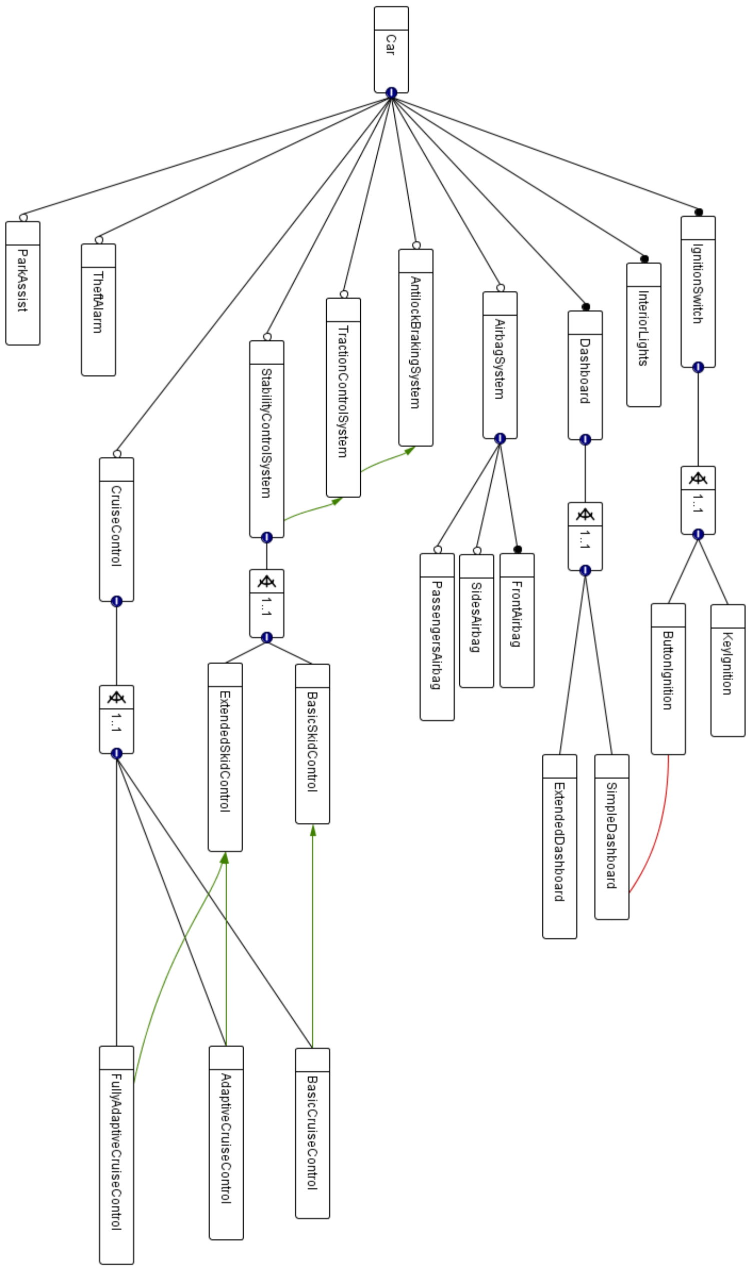


Figure 8.2: *Case study feature model*

the dashboard can be either a simple dashboard system or an extended dashboard system and also the stability control system can be either basic or extended. For the cruise control system, there are three possibilities: basic cruise control system, adaptive cruise control system and fully adaptive cruise control systems. For cars with an airbag system, the front airbags are compulsory. However, it can optionally have side airbags and/or passengers airbags. Green arrows in the figure demonstrate the dependencies between the features. In addition, red arrows show mutually exclusive features in our feature model (e.g. button ignition is not compatible with simple dashboard feature).

8.3.1 Products of SPL

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

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

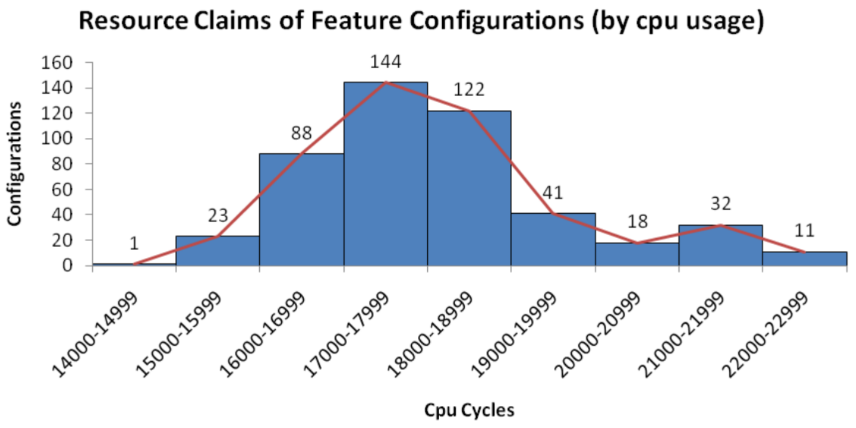


Figure 8.3: *Resource claims (CPU cycles) for all possible products in the feature model*

Car1

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

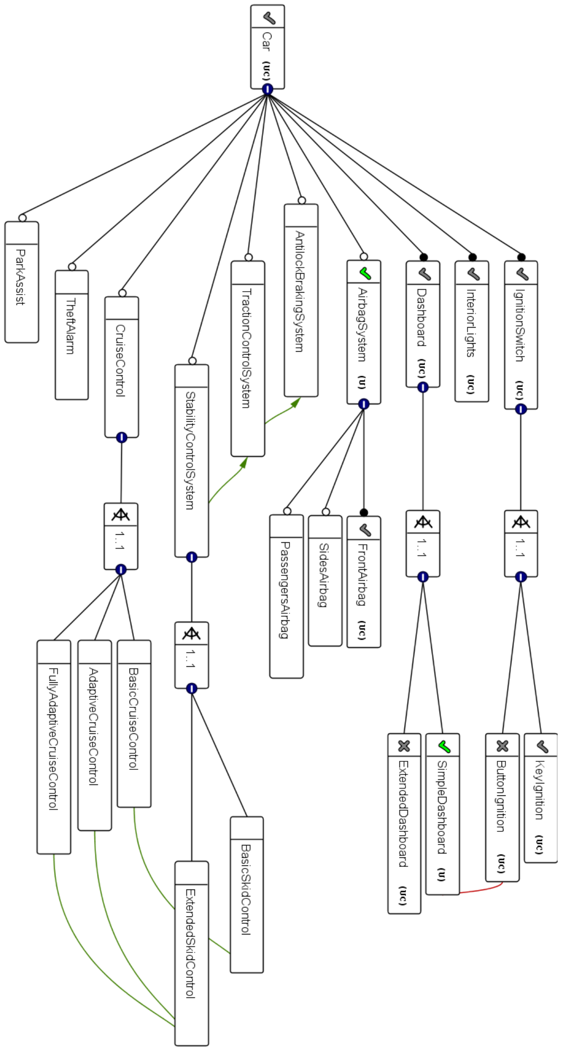


Figure 8.4: *Feature configuration for* Car1

Car2

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

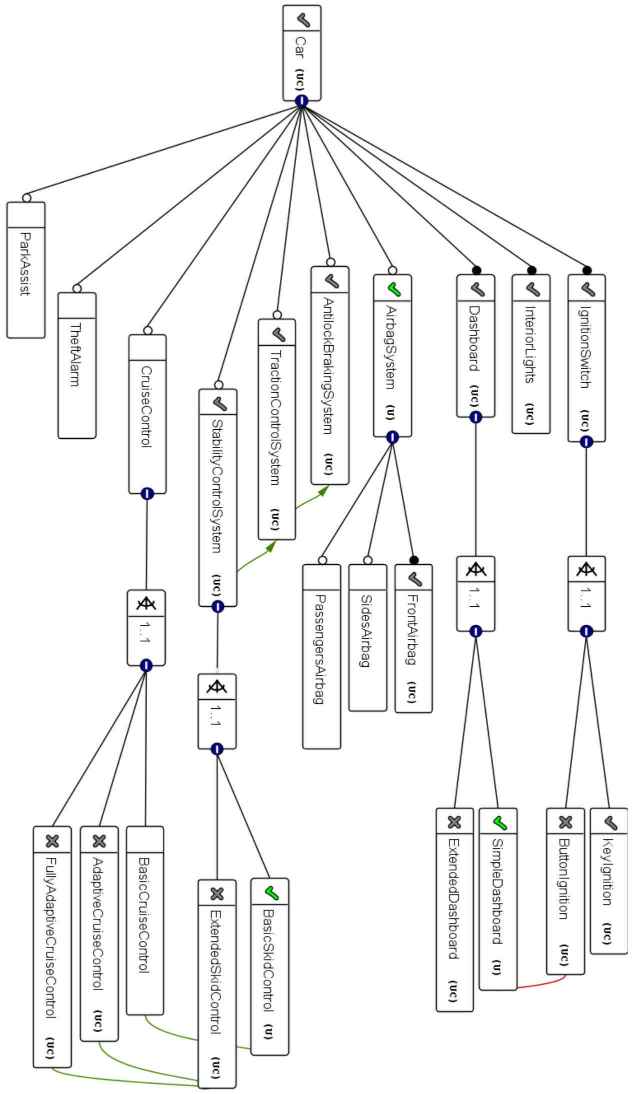


Figure 8.5: *Feature configuration for* Car2

Car3

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

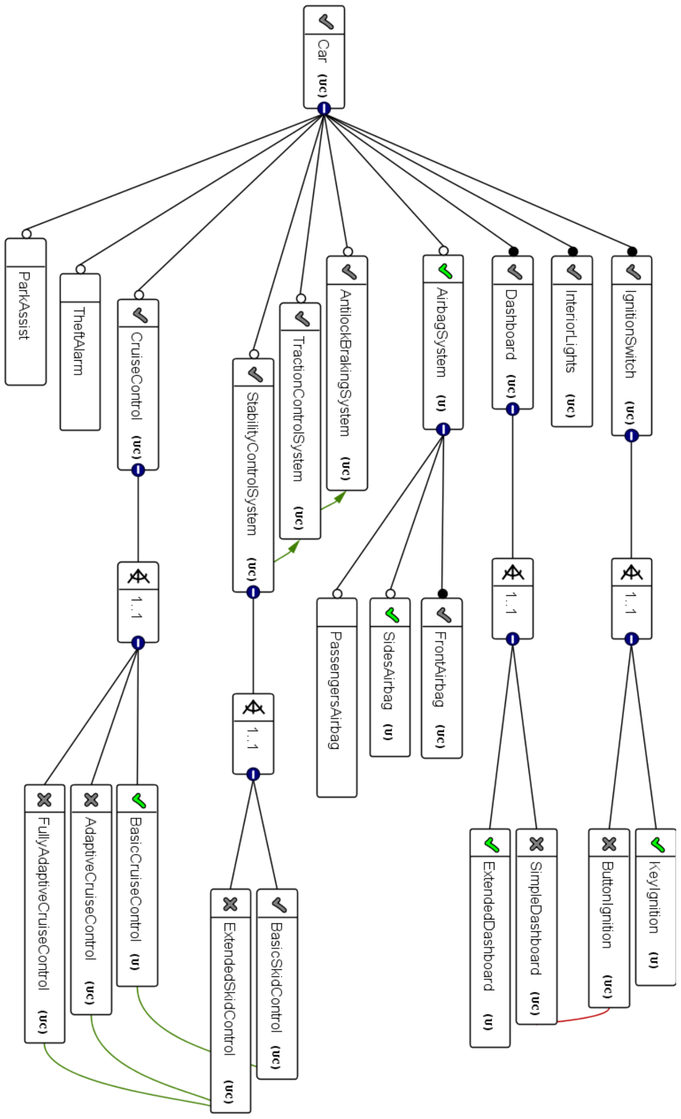


Figure 8.6: *Feature configuration for* Car3

Car4

Figure 8.7 depicts the feature configuration of *Car4*. In addition to the features of *Car3*, it has a button ignition and an adaptive cruise control system. It also provides the theft alarm functionality.

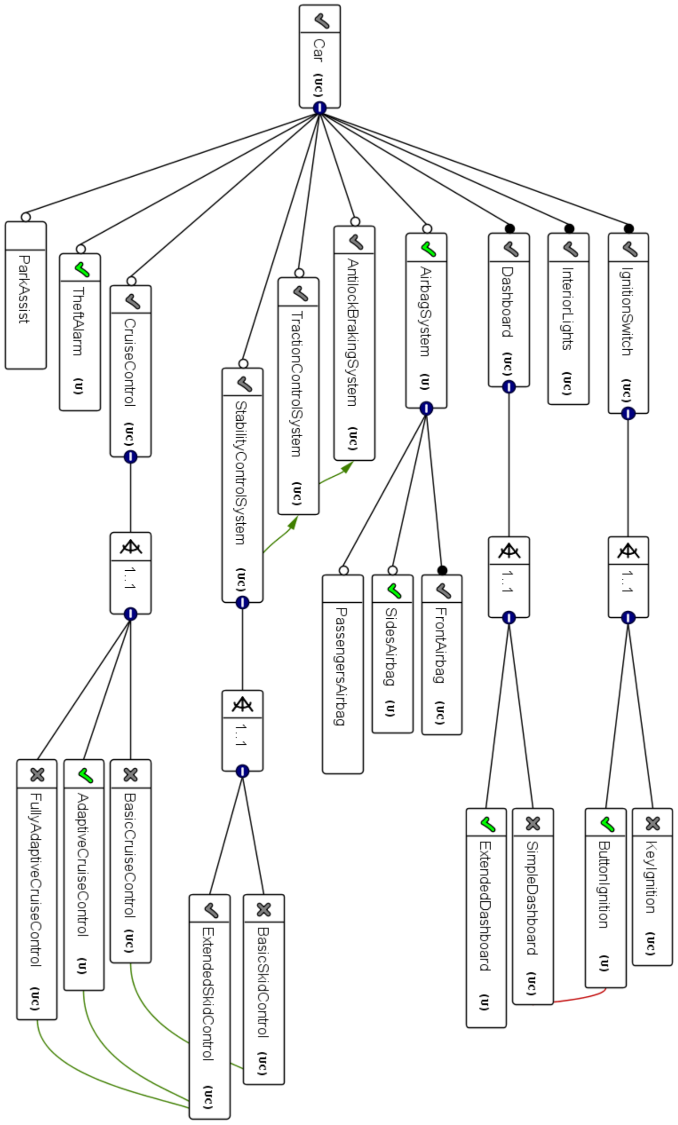


Figure 8.7: *Feature configuration for* Car4

Car5

Car5 (Figure 8.8) is the very high-end product in our product line. It has all the features (with the best one selected in mutually exclusive cases).

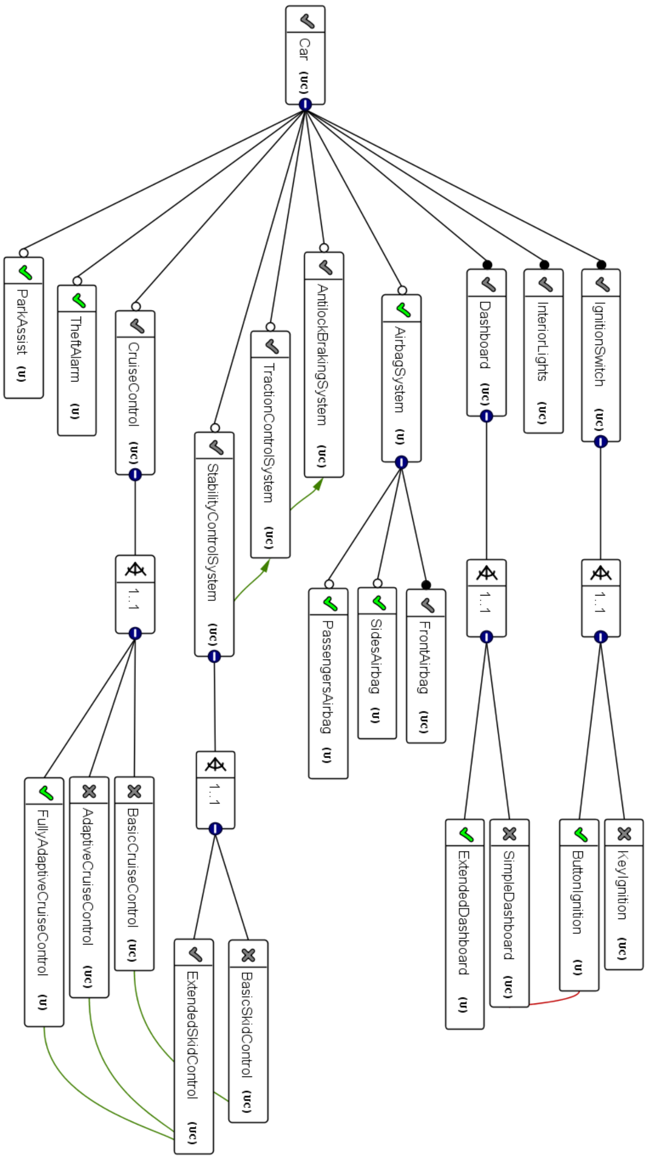


Figure 8.8: *Feature configuration for* Car5

8.3.2 Experiment Setup

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

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

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

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

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

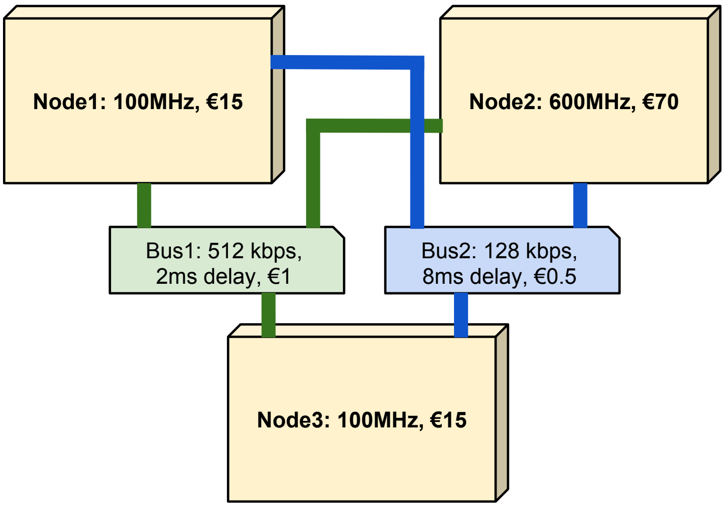
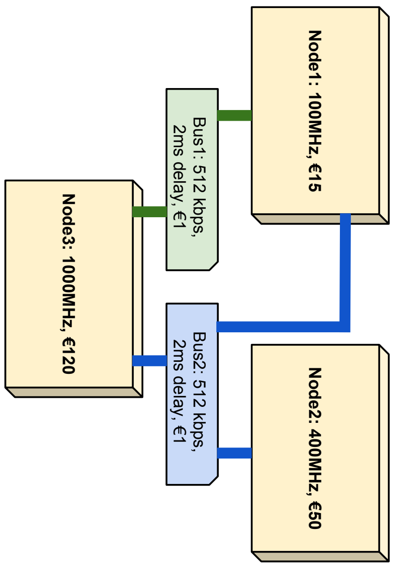
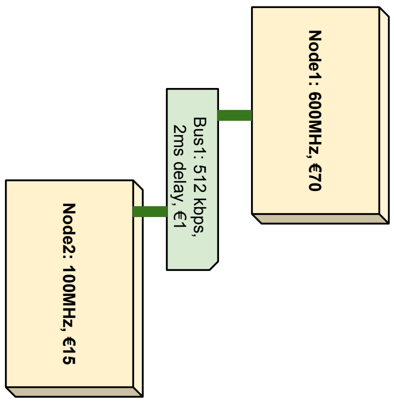

Node1: 100MHz, €15
Node2: 600MHz, €70
Bus1: 512 kbps, 2ms delay, €1
Bus2: 128 kbps, 8ms delay, €0.5
Node3: 100MHz, €15

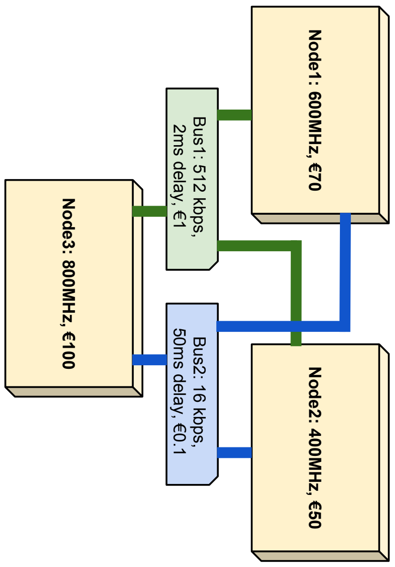
Figure 8.9: *Proposed common solution which is optimal for* Car1



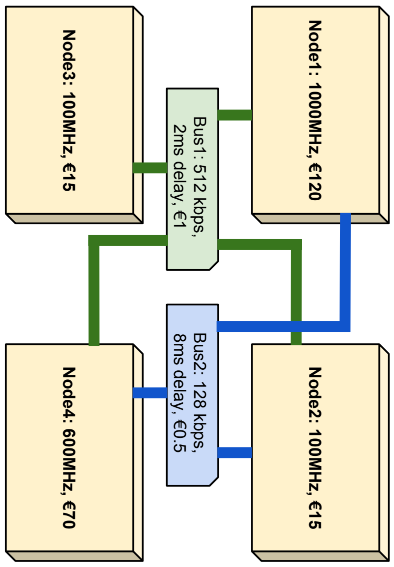
(a) *Similar solution which is optimal for* Car2



(b) *Similar solution which is optimal for* Car3


Node1: 600MHz, €70
Bus1: 512 kbps, 2ms delay, €1
Node3: 800MHz, €100
Bus2: 16 kbps, 50ms delay, €0.1
Node2: 400MHz, €50

(c) *Similar solution which is optimal for* Car4



(d) *Similar solution which is optimal for* Car5

Figure 8.10: *Similar solutions in other Pareto fronts*

# Solutions	*Δ = 2*	*Δ = 3*	*Δ = 4*	*Δ = 5*	*Δ = 6*	*Δ = 7*
Average	0.88	1.52	2.13	2.90	4.00	5.15

Table 8.1: *Average number of common solutions over 60 runs among various Pareto fronts*

8.3.3 Experiment Results

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

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

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

8.3.4 Validation

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

We applied our proposed method to an extended version of a case study based on an existing sub-system from the automotive industry. In the original version of the case study, we applied AQOSA to the same requirement specifications used by the architects at the automotive company when designing the current realization. In this chapter, we added new features to the case study like cruise control system, airbag system, park assist system, etc., to increase the complexity of the optimization problem. By doing this we ended up investigating a rather complex sub-system of realistic scale. Thus, the case study is partly based on real requirement specifications specifying a realistic sub-system that is relevant in an automotive context, and at the same time limited enough to explore and demonstrate how our search-based approach supports in solving this optimization problem. The resulting set of architecture solutions proposed by our method contains good candidates for manual assessment in later phases of the architecture design process for a product line. This is confirmed by comparing the set of architecture solutions to the existing sub-system architecture. However, the accuracy and relevance of the proposed architecture solutions are not the primary goals of this exploratory case study. Moreover, one of the advantages is that our approach can be executed as a distributed search approach which increases the speed of reaching to the results.

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

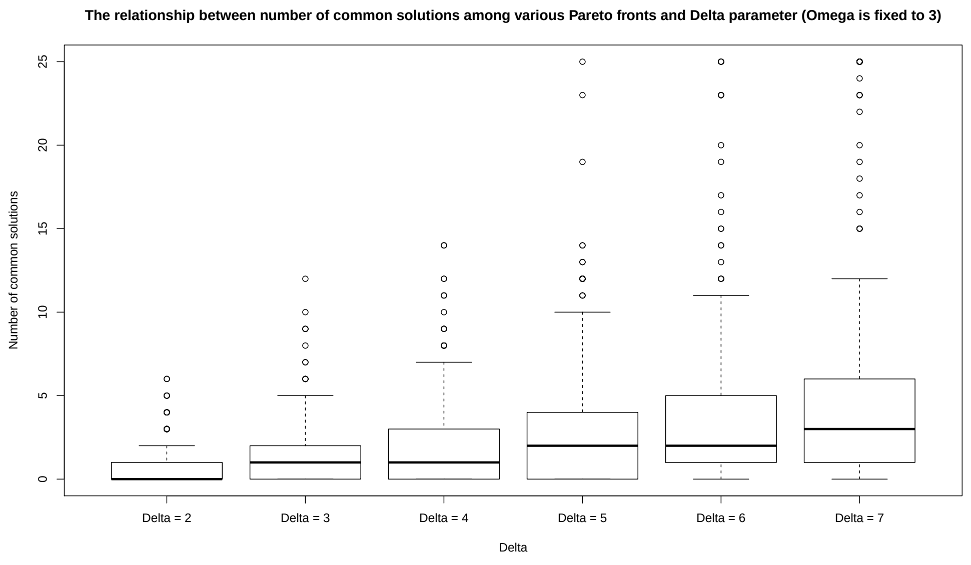


Figure 8.11: *Boxplot of the number of found common solutions for various* Δ

8.3.5 Interpretation of results

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

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

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

8.4 Summary

This chapter proposed a novel search-based method for finding optimal software architectural solutions which are applicable for a range of products in a product line. By introducing this approach, the AQOSA framework is extended to support multiple products at the same time. Earlier chapters discussed the AQOSA framework in a way that it aims at supporting architects in finding optimal architecture solutions in complex design situations with many potentially conflicting quality attributes. This chapter reported that AQOSA can be employed by architects in the context of software product lines as well.

We demonstrated the application of our proposed approach on an exploratory case study based on an existing sub-system from the automotive industry. The case study showed that while optimization techniques can find efficient solutions regarding all quality attributes, our method identifies similar optimal solutions that are applicable to the range of products in the software product line.

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

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

Chapter 9

Parallel Execution of Software Architecture Optimization

This chapter addresses the efficiency of the optimization algorithm (related to **RQ3** which is defined in Section 1.2):

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

In Chapter 7 we discussed efficiency improvements through using dedicated search operators. However, in this chapter we addresses **RQ3** from a complementary perspective: that of parallel execution. We know that meta-heuristic approaches in multi-objective problems, especially for high dimensions, mostly take long to execute. One of the best solutions to speed up this process is parallelising execution of evolutionary algorithm on multiple nodes of a super computer or in the cloud.

This chapter presents the results of parallelising execution of evolutionary algorithm for multi-objective optimization of software architecture. It reports on two different approaches for parallel execution of evolutionary algorithm: (1) a MapReduce approach [DG04], (2) an actor-based approach [HBS73].

This chapter is structured as follows. Firstly, Section 9.1 introduces the famous model for concurrency which is called MapReduce. MapReduce is inspired by functional programming constructs for processing (potentially large) lists of data. Then, Section 9.2 introduces a model of concurrency which is based on actors and messaging between them. These two sections also discuss two popular corresponding frameworks which implement those models: the Apache Hadoop is the most famous implementation of the MapReduce model; the Akka framework is an implementation for actor-based distribution. After that, Section 9.3 represents the results of an experiment where the parallel implementations of our proposed approach are studied (using our running case from the automotive industry). Finally, Section 9.4 summarizes this chapter.

9.1 The MapReduce Paradigm (with the Hadoop Framework)

MapReduce was firstly introduced by Dean et al. in [DG04]. MapReduce is a programming model designed for processing large volumes of data in parallel by dividing the work into a set of independent tasks. MapReduce programs are written in a particular style influenced by functional programming constructs, specifically idioms for processing lists of data. The processing of the list is distributed across a large number of machines operating in parallel. This model would not scale to large clusters if the components were allowed to share data arbitrarily. The communication overhead required to keep the data on the nodes synchronized at all times would prevent the system from performing reliably or efficiently at large scale. Users specify a map function that processes a key-value pair to generate a set of intermediate key-value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Programs written in this functional style are automatically parallelised. The resulting code can be executed on a large cluster of (commodity) machines.

The Apache Hadoop [Thea] software library is a framework that implements the MapReduce programming model. This framework allows for the distributed processing of large data sets across clusters of computers using the MapReduce paradigm. Conceptually, MapReduce programs transform (in parallel) lists of input data elements into lists of output data elements. Figure 9.1 shows a visualization of this process. A MapReduce program typically acts along the following lines:

1. Input data, such as a long text file, is split into key-value pairs. These key-value pairs are then fed to the mapper. (This is the job of Hadoop framework.)
2. The mapper processes each key-value pair individually and outputs one or more intermediate key-value pairs.
3. All intermediate key-value pairs are collected, sorted, and grouped by key (again, this step is automatically handled by the Hadoop framework).
4. For each unique key, the reducer receives the key with a list of all the values associated with it. The reducer aggregates these values in some way (adding them up, taking averages, finding the maximum, etc.) and outputs one or more output key-value pairs.
5. Output pairs are collected and stored in an output file (by the framework).

In this setting, the mapper function and the reduce function are the parts that can be programmed by the application developer.

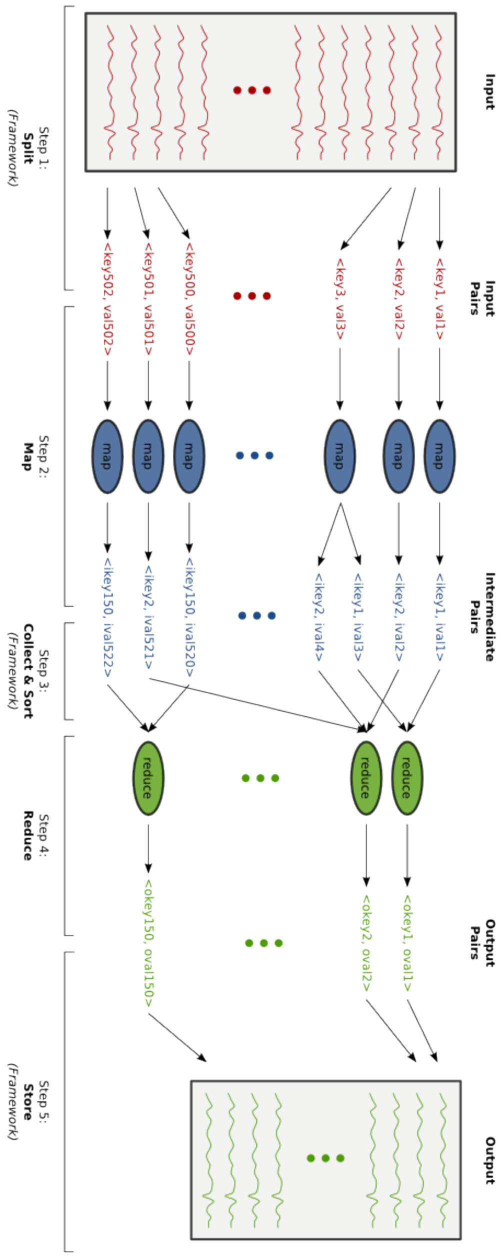


Figure 9.1: *A visualization of Map and Reduce processes*

Mapping List The first phase of a MapReduce program is called mapping. A list of data elements are provided, one at a time, to a function called the Mapper, which transforms each element individually to an output data element. As an example of the utility of map: Suppose you had a function toUpper(str) which returns an uppercase version of the input string. You could use this function with map to turn a list of strings into a list of uppercase strings. Note that we are not modifying the input string, we are returning a new string that will form part of a new output list.

Reducing List Reducing lets you combine values together. A reducer function iterates over the values of a list. It combines these values together, returning a single output value. Reducing is often used to produce "summary" data, turning a large volume of data into a smaller summary of itself. For example, "+" can be used as a reducing function, to return the sum of a list of input values. Examples of alternatives are *max*, *length*.

9.1.1 Case Study for the MapReduce Approach

We implemented MapReduce approach in the AQOSA framework and we run so many experiments with various parameters and settings. However, unfortunately we could not achieve parallelisation efficiency higher that 30% in none of these experiments with the famous Hadoop approach. Therefore, we decided to move on and try another approach which is described in the rest of this chapter.

9.2 Actor-based Distribution (with the Akka Framework)

The Actor Model provides a high level of abstraction for writing concurrent and distributed software application. It alleviates the developer from having to deal with explicit locking and thread management, making it easier to write correct concurrent and parallel systems. Actors were defined by Carl Hewitt [HBS73] but have been popularized by the Erlang language. Figure 9.2 depicts a simple model of actor-based concurrency where actors are represented as communicating event loops. The dotted lines represent the actor's event loop threads which perpetually take messages from their message queue and synchronously execute the corresponding methods on the actor's owned objects.

Actors give developers:

1. simple and high-level abstractions for concurrency and parallelism,

2. asynchronous, non-blocking and highly performant event-driven programming model,

3. very lightweight event-driven processes.

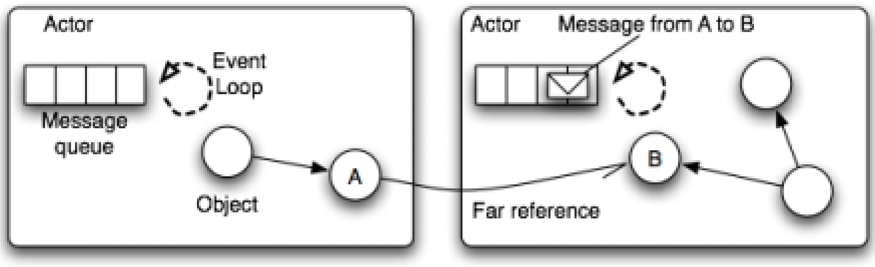


Figure 9.2: *Concurrency with actors and asynchronous message sending*

Akka [Akk] is an actor-based framework which helps developers in writing correct concurrent, fault-tolerant and scalable applications. Actors provide abstractions for transparent distribution and the basis for truly scalable and fault-tolerant applications.

9.3 Case Study for the Actor-based Approach

9.3.1 Implementation of the Actor-based Approach

Figure 9.3 depicts a schema of the actor-based Akka implementation of AQOSA. Five nodes were used: 1 master node and 4 worker nodes. The Akka framework was programmed to initialize 4 actors on each individual worker node. Hence, 16 actors were initialized in total. These worker-actors were responsible for evaluating an individual candidate solution based on predefined software quality attributes, such as response time, processor utilization, bus utilization, safety and cost. On the master node, the Akka framework was programmed to start one actor called 'Evaluator Balancer' (as depicted in Figure 9.3). This actor is responsible for distributing evaluation jobs to each of the 16 worker actors. This Evaluator-Balancer used a round-robin strategy for assigning jobs to workers. The AQOSA framework also ran on the master node and it invoked the Balancer-actor whenever it wanted to evaluate an individual candidate solution.

To examine the efficiency of the actor-based distributed implementation of our software architecture optimization framework, a new experiment was run for the SAAB Instrument Cluster case study again (see Section 5.3 for more details). This experiment was run on the DAS-4 [Theb] super computer. In this supercomputer every node is a powerful computer with a 8-core processor (each core runs at a speed of 2.67GHz and has 12MB cache) and 48GB memory.

9.3.2 Experiment Setup

For generating new architectural solutions, the repository of hardware components contained the following elements:

- 28 Processors: ranging over 14 various processing speeds from 66MHz to 500MHz; Each has two levels of failure rate. A processor is more expensive if it has less chance of failure.

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

After defining the above hardware options, AQOSA was run 30 times based while using the NSGA-II algorithm with the following parameter settings: initial population size(α) = 256, parent population size (μ) = 64, number of offspring(λ) = 64, archive size = 32, number of generations = 60, crossover rate set to 0.95, and all quality attributes are aimed to be minimized.

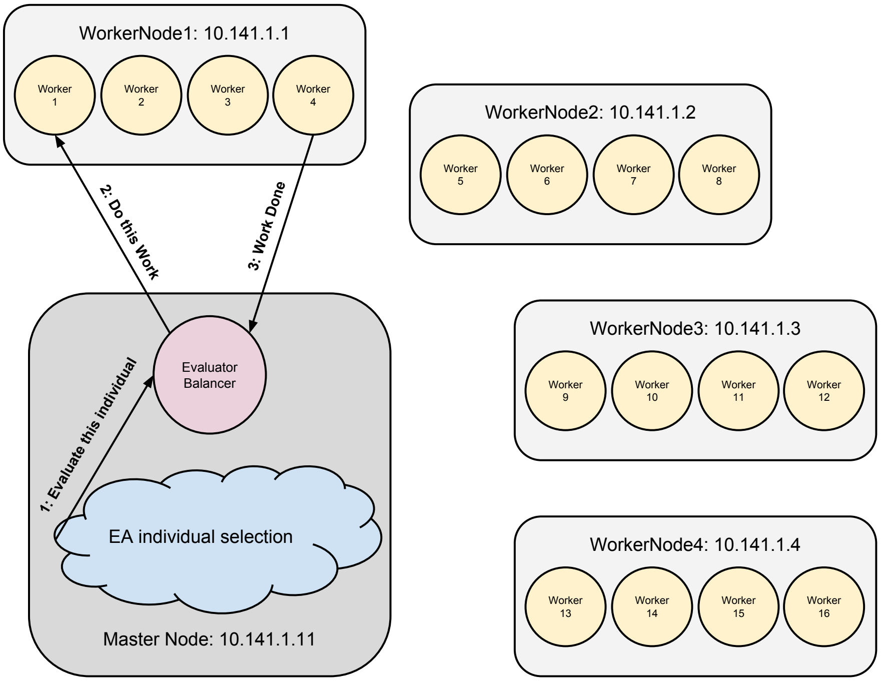


Figure 9.3: *AQOSA implementation of actor-based distribution scheme*

Run #	*Distributed (1 Master-Node + 4 Worker-Nodes)*	*Single-Node*
1	168,346	685,668
2	163,278	691,741
3	171,185	687,933
4	191,425	678,725
5	173,486	683,875
6	212,667	697,605
7	185,926	681,893
8	169,970	689,065
9	135,545	695,583
10	162,341	687,381
11	176,953	693,289
12	164,833	689,954
13	153,570	681,492
14	184,530	692,063
15	148,388	655,434
16	169,537	669,622
17	166,597	686,289
18	212,164	676,475
19	158,257	676,324
20	163,089	684,778
21	170,300	677,652
22	138,852	684,308
23	169,592	691,148
24	155,911	671,799
25	166,818	692,061
26	182,757	680,180
27	143,056	689,571
28	161,778	690,718
29	158,040	681,744
30	171,396	679,107
Average	168, 353	684, 116
Std. Deviation	17, 654	8, 766

Table 9.1: *Execution time (in ms) of 30 runs of experiment*

9.3.3 Experiment Results

Table 9.1 shows the execution times (in milliseconds) of 30 runs of the experiment. The first column is the execution number. The second column is the execution times of the actor-based distributed implementation. As described in Section 9.3.1, the application was distributed to 1 master node and 4 worker nodes. The third column shows the execution times of running the same design problem on single node.

In parallel computing the speedup is used as a measure of the improvement obtained by of parallelising a computation. For a system with p processors, speedup is defined as:

$$S_p = \frac{T_1}{T_p}. \tag{9.1}$$

where T_1 is the execution time of the sequential algorithm, and T_p is the execution time of the parallel algorithm using p processors. Therefore, in our experiment the speedup for the average over 30 runs is:

$$S_5 = \frac{684,116}{168,353} = 4.0635 \tag{9.2}$$

Additionally, efficiency of a parallel algorithm is defined by the following formula:

$$E_p = \frac{S_p}{p} = \frac{T_1}{p \times T_p}. \tag{9.3}$$

To calculate the efficiency of our actor-based distributed implementation of the optimization, the aforementioned formula is applied:

$$E_5 = \frac{S_5}{5} = \frac{4.0635}{5} = 0.8127 \tag{9.4}$$

In other words, our actor-based distributed implementation in case of this experiment on a real-world case study shows 81.27% efficiency. This number indicates a good efficiency hence suggests that this is an acceptable approach for the parallelisation of the optimization.

9.4 Summary

This chapter presented the results of different strategies for parallel execution of our evolutionary optimization approach. The experiment was defined based on an industrial case study and was applied to a software architecture optimization problem with five objectives. The achieved results showed that parallel execution of evolutionary algorithm for software architecture optimization can improve execution time significantly with acceptable efficiency in multi-objective optimization context.

The results show that for cases in which the evaluation calculation takes significantly more time compared to the selection calculation (of new candidate solutions), the efficiency of parallelisation is considerable. However, for cases in which the evaluation process is fast, parallelisation may not help considerably. When comparing the actor-based approach and the MapReduce approach, at least in our case study, shows that the actor-based approach shows better speedup.

Chapter 10

Conclusion

In this work we have presented a meta-heuristic optimization approach for automated software architecture design. We have studied 4 research questions in this dissertation. This chapter summaries the findings of this study.

This chapter is structured as follows. Section 10.1 presents our findings with regards to each research question of the dissertation. Section 10.2 proposes some directions for future work in this research area.

10.1 Summary of Findings

In this dissertation we described a meta-heuristic optimization approach for automated software architecture design and its associated tooling that we developed. This approach offers a new tool to aid architects in finding good designs in complex design situations with potentially conflicting multiple quality requirements. Furthermore, the tool reduces the development time and improves the quality of the architecture design. The AQOSA framework supports multiple quality attributes for optimization including response time, processor utilization, bus utilization, safety, and cost. As such it supports more quality properties than any other automated software architecture design tool.

Inspired by the model-driven approach, the framework uses a single integrated representation of the software architecture (AQOSA IR) which is the basis for performing multiple quality analysis based. Moreover, this framework is designed so that it can be easily extended with additional quality attributes.

The approach has been applied on industrial case studies with the aim of finding better solutions than the current realization while fulfilling the same requirements and constraints.

The use of the AQOSA framework improves over the state of the art because:

- AQOSA is modelling language independent. It can interoperate with various architectural modelling languages, in particular UML, AADL. Because AQOSA is based on the component-based paradigm but makes only few assumptions on the component model used.

- AQOSA supports multiple degrees of freedom for automatically generating alternative architectures. In particular, we developed an approach to support variation of the hardware topology as a new degree of freedom.

- AQOSA optimizes multiple quality attributes at once. To the best of our knowledge AQOSA is the first approach which supports evaluation and optimization of five quality attributes simultaneously.

In the following sections, the conclusions collected throughout this dissertation are summarized and used to address the research questions central to this dissertation (Section 1.2).

10.1.1 Research Question 1

RQ1 : Can meta-heuristic optimization improve the process of designing efficient architectures for a given set of quality attributes in an industrial domain?

The case studies in this dissertation demonstrated the usefulness of the AQOSA software architecture optimization framework. These case studies range from business information systems to embedded systems in the automotive industry.

The main case in the dissertation reported on a real-world large-scale industrial case study applying a meta-heuristic optimization approach for automated software architecture design which supports multiple quality attributes. The case showed in an industrial context how meta-heuristic optimization approaches can improve software architecture design with respect to multiple quality attributes, and could suggest a wide range of optimized architectural solutions. Comparing the solutions proposed by AQOSA to the existing realization showed that AQOSA is able to synthesize solutions that are efficient in all quality attributes while fulfilling given constraints. Also, in contrast to human architects who tend to propose solutions based on previous architectures, AQOSA proposes revolutionary solutions. This revolutionary suggestions may open new possibilities to architects.

Although the case study shows the saving of manual effort which is beneficial for time-to-market and development cost, it also shows that the proposed architecture solutions needs to be assessed by human architects. Hence, this dissertation demonstrated how an architecture design framework like AQOSA complements the domain knowledge and experience of the architect, rather than replaces the architect.

The case studies showed that evolutionary multi-objective optimization (EMO) can improve the process of designing efficient architectures for a set of given quality attributes in the following aspects:

- EMO can efficient search through large solution spaces,
- EMO can consider multiple system quality attributes in design and analysis,
- EMO can proposes revolutionary solutions.

10.1.2 Research Question 2

RQ2 : Can enlargement of the optimization search space help the meta-heuristic approach to find better architectural solutions?

To address the enlargement of the search space, in this dissertation two novel degrees of freedom were introduced:

1. a new method for varying the topology of software architectures,
2. allowing architecture to replicate software component instances.

Introducing more degrees of freedom essentially enlarges the solution space, and thus the search space. If this enlargement is 'in the right direction', then it allows evolutionary algorithms to find better solutions. We could show by running a very computationally-intensive experiment on an industrial case study, that optimization using this approach indeed finds better software architectures. These experiments bring empirical evidences that show that better solutions can be found by using these new degrees of freedom.

10.1.3 Research Question 3

RQ3 : In which ways can meta-heuristic optimization be improved in order to make the process of reaching optimal architectural solutions faster?

To answer this research question, we consider two different points of view:
Firstly, in this dissertation: (i) the usefulness of problem-specific operators in the software architecture domain was discussed, and (ii) a comparison between various approaches for combing heuristic-based search operators was performed. To do so, knowledge of architecture anti-patterns was implemented by means of problem-specific search operators within an evolutionary algorithm. The results of the case study experiment showed that search operators for improving one objective can be used in multi-objective optimization context. The results indicated that proper combination strategies for heuristic-based search operators can lead optimization algorithms to

optimal solutions faster. However, in order to prevent getting trapped in suboptimal solutions, randomness should always be included in the optimization, especially in the offspring mating process.

Secondly, the dissertation presented the results of two different strategies for parallel execution of our evolutionary optimization approach. The achieved results showed that parallel execution of evolutionary algorithm for software architecture optimization can improve execution time significantly with acceptable efficiency in multi-objective optimization context. The results show that for cases in which the evaluation calculation takes significantly more time compared to the selection calculation (of new candidate solutions), the efficiency of parallelization is considerable. When performed a comparison between an actor-based approach and a MapReduce approach to parallelizing the AQOSA computations. In our case study, the actor-based approach shows the better speedup of these two. This is probably due to the synchronization policy of the MapReduce paradigm that does not match well with the concurrent evaluation tasks that vary a lot in computation time.

10.1.4 Research Question 4

RQ4 : In what aspects can search-based approaches improve the process of designing a software architecture for a family of products in a software product line?

For this research question, this dissertation proposes a novel search-based method for finding optimal software architectural solutions which are applicable for a range of products in a product line. To achieve this, we extended the AQOSA framework to support multiple products at the same time. Moreover, we introduced a notion of *"distance"* between solutions that is an indicator for the number of changes that need to be made to create one architecture out of another. We demonstrated the application of our proposed approach on an exploratory case study based on an existing sub-system from the automotive industry. The case study showed that our method identifies similar optimal solutions that are applicable to the range of products in the software product line.

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

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

10.2 Future Work

One interesting direction for the future work is interactive search-based approaches. By the means of interactive search, the architect would be able to guide optimization process interactively to a specific solution space area. In this way, search is steered jointly by architect preferences and software optimization algorithm.

As future work regarding the heuristic-based search operators, it is interesting to study situations with an unbalanced number of operators which each change different objectives. For example, 3 operators in favour of one objective and 2 operators in favour of some conflicting objectives. Also, another topic for future work can be studying effects of adding weights to heuristic-based search operators on the results of the optimization process.

For future work in the area of software architecture optimization for multiple products, we suggest the following two directions: Firstly, this approach can be improved in the optimization process by using a co-evolving Pareto fronts technique [KCV02]. Hence, the algorithm can switch some of the archive solutions from one population to the another one, every few generations. By doing that, it might be the case that the optimization process gives us better results in terms of commonality with other Pareto fronts. Secondly, it would be interesting to investigate the results of employing a search-based approach for exploring feature combinations. We know large feature models lead up to million of possibilities for feature combinations. For those situations, going one by one through all of the configurations is not a feasible approach and a search-based approach is needed for that exploration as well. In other words, it would be a complex two-level optimization problem: both at the feature-level and at the architecture-level.

Regarding the future work in the parallel execution, it is interesting to extend the parallelization to include the selection step of the evolutionary algorithm as well. In this way, in addition to the evaluation process, also the selection algorithm could execute in parallel which helps efficiency of parallelization even more.

Appendix A

AQOSA IR Sample Model

In this appendix, a sample of AQOSA IR (described in Chapter 4.3) is presented. Listing A.1 shows the source of the AQOSA IR model for the case study mentioned in Section 5.3. It is encoded in the Eclipse EMF [Ecl] format. This model is also accessible via this URL: http://goo.gl/nio8u8.

Listing A.1: *AQOSA IR EMF model for Saab Instrument Cluster system*

```xml
<?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>
```

```xml
    <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"/>
      <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="10000.0" missedPercentage="0.05">
      <flowinstance instance="//@assembly/@flow.0" start="300.0" trigger="5000.0"
           deadline="75.0"/>
      <flowinstance instance="//@assembly/@flow.1" start="400.0" trigger="5000.0"
           deadline="75.0"/>
      <flowinstance instance="//@assembly/@flow.2" start="500.0" trigger="5000.0"
           deadline="50.0"/>
      <flowinstance instance="//@assembly/@flow.3" start="1000.0" trigger="100.0"
           deadline="50.0"/>
      <flowinstance instance="//@assembly/@flow.4" start="1030.0" trigger="100.0"
           deadline="35.0"/>
      <flowinstance instance="//@assembly/@flow.5" start="100.0" trigger="1000.0"
           deadline="50.0"/>
      <flowinstance instance="//@assembly/@flow.6" start="1070.0" trigger="100.0"
           deadline="30.0"/>
      <flowinstance instance="//@assembly/@flow.7" start="2000.0" trigger="5000.0"
           deadline="250.0"/>
      <flowinstance instance="//@assembly/@flow.8" start="2500.0" trigger="5000.0"
           deadline="250.0"/>
      <flowinstance instance="//@assembly/@flow.9" start="3000.0" trigger="5000.0"
           deadline="125.0"/>
    </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" 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"/>
    <bus id="CAN-HS" bandwidth="500.0" delay="0.002" cost="100.0"/>
    <bus id="CAN-MS" bandwidth="125.0" delay="0.008" 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="3" noSampling="50" noDuplicate="1" minCost="200.0" maxCost="10000.0"
         >
      <evaluations>ResponseTime</evaluations>
      <evaluations>CPUUtilization</evaluations>
      <evaluations>BusUtilization</evaluations>
      <evaluations>Safety</evaluations>
      <evaluations>Cost</evaluations>
    </settings>
  </objectives>
</aqosa.ir:AQOSAModel>
```

List of Figures

List of Tables

Bibliography

[ABG$^+$14] Aldeida Aleti, Barbora Buhnova, Lars Grunske, Anne Koziolek, and Indika Meedeniya. Software architecture optimization methods: A systematic literature review. In Wilhelm Hasselbring and Nils Christian Ehmke, editors, *Software Engineering*, volume 227 of *LNI*, pages 77–78. GI, 2014. URL: http://eprints.uni-kiel.de/23752/. (cited on page 99).

[ABGM09] Aldeida Aleti, Stefan Björnander, Lars Grunske, and Indika Meedeniya. ArcheOpterix: An extendable tool for architecture optimization of AADL models. In *MOMPES*, pages 61–71. IEEE Computer Society, 2009. doi:10.1109/MOMPES.2009.5069138. (cited on page 18).

[ABM$^+$13] Michal Antkiewicz, Kacper Bak, Alexandr Murashkin, Rafael Olaechea, Jia Hui (Jimmy) Liang, and Krzysztof Czarnecki. Clafer tools for product line engineering. In *SPLC Workshops*, pages 130–135. ACM, 2013. doi:10.1145/2499777.2499779. (cited on page 117).

[AGHIR12] Silvia Abrahão, Javier Gonzalez-Huerta, Emilio Insfrán, and Isidro Ramos. Software evolution in model-driven product line engineering. *ERCIM News*, 2012(88), 2012. (cited on page 24).

[Akk] Akka Framework: an open-source toolkit and runtime simplifying the construction of concurrent and distributed applications on the JVM. http://akka.io/. URL: http://akka.io/. (cited on page 139).

[ALBG99] Daniel Amyot, Luigi Logrippo, Raymond J. A. Buhr, and Tom Gray. Use case maps for the capture and validation of distributed systems requirements. In *RE*, page 44. IEEE Computer Society, 1999. doi:10.1109/ISRE.1999.777984. (cited on page 56).

[Axe09] Jakob Axelsson. Evolutionary architecting of embedded automotive product lines: An industrial case study. In *WICSA/ECSA*, pages 101–110. IEEE, 2009. doi:10.1109/WICSA.2009.5290796. (cited on page 82).

[Bäc96] Thomas Bäck. *Evolutionary algorithms in theory and practice - evolution strategies, evolutionary programming, genetic algorithms*. Oxford University Press, 1996. URL: http://dl.acm.org/citation.cfm?id=229867. (cited on pages 7 and 8).

[BBL10] Michael Bowman, Lionel C. Briand, and Yvan Labiche. Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms. *IEEE Trans. Software Eng.*, 36(6):817–837, 2010. doi:10.1109/TSE.2010.70. (cited on page 22).

[BCdW06] Egor Bondarev, Michel R. V. Chaudron, and Peter H. N. de With. A Process for Resolving Performance Trade-Offs in Component-Based Architectures. In Ian Gorton and et al., editors, *CBSE*, volume 4063 of *LNCS*, pages 254–269. Springer, 2006. doi:10.1007/11783565_18. (cited on pages 17 and 27).

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003. URL: http://dl.acm.org/citation.cfm?id=773239. (cited on page 14).

[BKLW95] Mario Barbacci, Mark H. Klein, Thomas A. Longstaff, and Charles B. Weinstock. Quality Attributes. Technical Report CMU/SEI-95-TR-021, Carnegie Mellon, 1995. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=12433. (cited on pages 14 and 43).

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009. doi:10.1016/j.jss.2008.03.066. (cited on pages 13, 17 and 27).

[BNE07] Nicola Beume, Boris Naujoks, and Michael Emmerich. Sms-emoa: Multiobjective selection based on dominated hypervolume. *European Journal of Operational Research*, 181(3):1653–1669, 2007. doi:10.1016/j.ejor.2006.08.008. (cited on pages 34 and 52).

[BR03] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308, 2003. doi:10.1145/937503.937505. (cited on page 2).

[BS14] Maxim Buzdalov and Anatoly Shalyto. A provably asymptotically fast version of the generalized jensen algorithm for non-dominated sorting. In Thomas Bartz-Beielstein, Jürgen Branke, Bogdan Filipic, and Jim Smith, editors, *Parallel Problem Solving from Nature - PPSN XIII - 13th International Conference, Ljubljana, Slovenia, September 13-17, 2014. Proceedings*, volume 8672 of *Lecture Notes in Computer Science*, pages 528–537. Springer, 2014. doi:10.1007/978-3-319-10762-2_52. (cited on page 48).

[CFD+11] Andrea Ciancone, Antonio Filieri, Mauro Luigi Drago, Raffaela Mirandola, and Vincenzo Grassi. Klapersuite: An integrated model-driven environment for reliability and performance analysis of component-based systems. In Judith Bishop and Antonio Vallecillo, editors, *TOOLS (49)*, volume 6705 of *LNCS*, pages 99–114. Springer, 2011. doi:10.1007/978-3-642-21952-8_9. (cited on page 19).

[CFL+10] Marco Ceriani, Fabrizio Ferrandi, Pier Luca Lanzi, Donatella Sciuto, and Antonino Tumeo. Multiprocessor systems-on-chip synthesis using multi-objective evolutionary computation. In Martin Pelikan and Jürgen Branke, editors, *Genetic and Evolutionary Computation Conference, GECCO 2010, Proceedings, Portland, Oregon, USA, July 7-11, 2010*, pages 1267–1274. ACM, 2010. doi:10.1145/1830483.1830710. (cited on page 20).

[Clo] CloudBees. CloudBees: The Java PaaS (Platform-as-a-Service). URL: http://www.cloudbees.com/. (cited on page 47).

[CMT10] Vittorio Cortellessa, Antinisca Di Marco, and Catia Trubiani. Performance antipatterns as logical predicates. In Radu Calinescu, Richard F. Paige, and Marta Z. Kwiatkowska, editors, *ICECCS*, pages 146–156. IEEE Computer Society, 2010. doi:10.1109/ICECCS.2010.44. (cited on page 102).

[CN01] Paul C. Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. URL: http://dl.acm.org/citation.cfm?id=501065. (cited on page 2).

[Col12] Thelma Elita Colanzi. Search based design of software product lines architectures. In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *ICSE*, pages 1507–1510. IEEE, 2012. doi:10.1109/ICSE.2012.6227050. (cited on page 23).

[CPSV08] Vittorio Cortellessa, Pierluigi Pierini, Romina Spalazzese, and Alessio Vianale. MOSES: MOdeling Software and platform architEcture in UML 2 for Simulation-based performance analysis. In Steffen Becker, Frantisek Plasil, and Ralf Reussner, editors, *QoSA*, volume 5281 of *LNCS*, pages 86–102. Springer, 2008. doi:10.1007/978-3-540-87879-7_6. (cited on page 19).

[CV12] Thelma Elita Colanzi and Silvia Regina Vergilio. Applying search based optimization to software product line architectures: Lessons learned. In Gordon Fraser and Jerffeson Teixeira de Souza, editors, *SSBSE*, volume 7515 of *Lecture Notes in Computer Science*, pages 259–266. Springer, 2012. doi:10.1007/978-3-642-33119-0_19. (cited on page 23).

[DAPM02] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002. doi:10.1109/4235.996017. (cited on pages 33 and 52).

[Depa] Department of Computer Science at the University of Erlangen-Nuremberg. Opt4J: A Modular Framework for Meta-heuristic Optimization. URL: http://opt4j.sourceforge.net/. (cited on pages 33 and 47).

[Depb] Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology. Alloy: a language and tool for relational models. URL: http://alloy.mit.edu/alloy/index.html. (cited on page 118).

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In Eric A. Brewer and Peter Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150. USENIX Association, 2004. URL: http://www.usenix.org/events/osdi04/tech/dean.html. (cited on pages 135 and 136).

[DW00] Stefan Droste and Dirk Wiesmann. Metric based evolutionary algorithms. In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, editors, *EuroGP*, volume 1802 of *Lecture Notes in Computer Science*, pages 29–43. Springer, 2000. URL: http://dl.acm.org/citation.cfm?id=646808.703953. (cited on page 100).

[EBN05] Michael Emmerich, Nicola Beume, and Boris Naujoks. An EMO algorithm using the hypervolume measure as selection criterion. In Carlos A. Coello Coello, Arturo Hernández Aguirre, and Eckart Zitzler, editors, *Evolutionary Multi-Criterion Optimization, Third International Conference, EMO 2005, Guanajuato, Mexico, March 9-11, 2005, Proceedings*, volume 3410 of *Lecture Notes in Computer Science*, pages 62–76. Springer, 2005. doi:10.1007/978-3-540-31880-4_5. (cited on page 34).

[Ecl] Eclipse open source community and Eclipse Foundation. Eclipse Modeling Framework (EMF). URL: http://www.eclipse.org/modeling/emf/. (cited on pages 30 and 151).

[EF11] Michael T. M. Emmerich and Carlos M. Fonseca. Computing hypervolume contributions in low dimensions: Asymptotically optimal algorithm and complexity results. In Ricardo H. C. Takahashi, Kalyanmoy Deb, Elizabeth F. Wanner, and Salvatore Greco, editors, *Evolutionary Multi-Criterion Optimization - 6th International Conference, EMO 2011, Ouro Preto, Brazil, April 5-8, 2011. Proceedings*, volume 6576 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2011. doi:10.1007/978-3-642-19893-9_9. (cited on page 48).

[EG13] Ulrik Eklund and Håkan Gustavsson. Architecting automotive product lines: Industrial practice. *Sci. Comput. Program.*, 78(12):2347–2359, 2013. doi:10.1016/j.scico.2012.06.008. (cited on page 120).

[EGS01] Michael Emmerich, Monika Grötzner, and Martin Schütz. Design of graph-based evolutionary algorithms: A case study for chemical process networks. *Evolutionary Computation*, 9(3):329–354, 2001. doi:10.1162/106365601750406028. (cited on page 100).

[EM08] Leire Etxeberria and Goiuria Sagardui Mendieta. Variability driven quality evaluation in software product lines. In *SPLC*, pages 243–252. IEEE Computer Society, 2008. doi:10.1109/SPLC.2008.37. (cited on page 23).

[FGH06] Peter H. Feiler, David Gluch, and John J. Hudak. The Architecture Analysis & Design Language (AADL): An Introduction. Technical Report CMU/SEI-2006-TN-011, Carnegie Mellon, 2006. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=7879. (cited on pages 18 and 27).

[Fie10] Tony Field. *JINQS: An Extensible Library for Simulating Multiclass Queueing Networks*, 2010. URL: http://www.doc.ic.ac.uk/~ajf/Research/manual.pdf. (cited on page 43).

[FS10] Marc Förster and Daniel Schneider. Flexible, Any-Time Fault Tree Analysis with Component Logic Models. In *ISSRE*, pages 51–60. IEEE Computer Society, 2010. doi:10.1109/ISSRE.2010.47. (cited on page 41).

[FT09] Marc Förster and Mario Trapp. Fault Tree Analysis of Software-Controlled Component Systems Based on Second-Order Probabilities. In *ISSRE*, pages 146–154. IEEE Computer Society, 2009. doi:10.1109/ISSRE.2009.22. (cited on page 44).

[GH11] Javier Gonzalez-Huerta. Integration of Quality Attributes in Software Product Line Development: Master en Ingenieria del Software Metodos Formales y Sistemas de Informacion. Master's thesis, UPV, Spain, 2011. (cited on pages 15, 55 and 89).

[GLHT10] Michael Glaß, Martin Lukasiewycz, Christian Haubelt, and Jürgen Teich. Lifetime Reliability Optimization for Embedded Systems: A System-Level Approach. In *Proceedings of RASDAT '10*, pages 17–22, 2010. URL: https://www12.informatik.uni-erlangen.de/publications/pub2010/GLHT10.pdf. (cited on page 20).

[Gooa] Google Inc. Google App Engine: Platform as a Service (PaaS) cloud computing platform for developing and hosting web applications in Google-managed data centers. URL: https://developers.google.com/appengine/. (cited on page 47).

[Goob] Google Inc. Google Chart: rich gallery of interactive charts and data tools which are powerful, simple to use. URL: https://developers.google.com/chart/. (cited on page 47).

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In Nils J. Nilsson, editor, *IJCAI*, pages 235–245. William Kaufmann, 1973. URL: http://dl.acm.org/citation.cfm?id=1624775.1624804. (cited on pages 135 and 138).

[HC01] George T. Heineman and William T. Councill, editors. *Component-based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. URL: http://dl.acm.org/citation.cfm?id=379381. (cited on page 13).

[HE13] Iris Hupkens and Michael Emmerich. Logarithmic-time updates in sms-emoa and hypervolume-based archiving. In *EVOLVE-A Bridge between Probability, Set Oriented Numerics, and Evolutionary Computation IV*, pages 155–169. Springer International Publishing, 2013. (cited on page 48).

[Hei12] Werner Heijstek. *Architecture Design in Global and Model-Centric Software Development*. PhD thesis, Leiden Institute of Advanced Computer Science (LIACS), Faculty of Science, Leiden University, 2012. URL: https://openaccess.leidenuniv.nl/handle/1887/20225. (cited on page 11).

[HF07] John J. Hudak and Peter H. Feiler. Developing AADL Models for Control Systems: A Practitioner's Guide. Technical Report CMU/SEI-2007-TR-014, Carnegie Mellon University, 2007. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=8437. (cited on pages 55 and 89).

[HHRV11] Ábel Hegedüs, Ákos Horváth, István Ráth, and Dániel Varró. A model-driven framework for guided design space exploration. In Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors, *ASE*, pages 173–182. IEEE, 2011. doi:10.1109/ASE.2011.6100051. (cited on page 19).

[HKP10] George T. Heineman, Jan Kofron, and Frantisek Plasil, editors. *Research into Practice - Reality and Gaps, 6th International Conference on the Quality of Software Architectures, QoSA 2010, Prague, Czech Republic, June 23 - 25, 2010. Proceedings*, volume 6093 of *Lecture Notes in Computer Science*. Springer, 2010. doi:10.1007/978-3-642-13821-8. (cited on page 176).

[Inf03] Information Technology for European Advancement (ITEA) Project. Robust Open Component Based Software Architecture for Configurable Devices Project (ROBOCOP), 2003. URL: http://www.hitech-projects.com/euprojects/robocop/. (cited on page 27).

[ISO01] ISO. ISO/IEC 9126-1:2001, Software engineering – Product quality – Part 1: Quality model. Technical report, International Organization for Standardization, 2001. URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22749. (cited on page 14).

[ISO11] ISO/IEC/IEEE. Systems and software engineering – architecture description. *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)*, pages 1 –46, 1 2011. doi:10.1109/IEEESTD.2011.6129467. (cited on page 11).

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=11231. (cited on page 15).

[KCV02] Nattavut Keerativuttitumrong, Nachol Chaiyaratana, and Vara Varavithya. Multi-objective co-operative co-evolutionary genetic algorithm. In Juan J. Merelo Guervós, Panagiotis Adamidis, Hans-Georg Beyer, José Luis Fernández-Villacañas Martín, and Hans-Paul Schwefel, editors, *PPSN*, volume 2439 of *Lecture Notes in Computer Science*, pages 288–297. Springer, 2002. doi:10.1007/3-540-45712-7_28. (cited on page 149).

[KKR11] Anne Koziolek, Heiko Koziolek, and Ralf Reussner. Peropteryx: automated application of tactics in multi-objective software architecture optimization. In Ivica Crnkovic, Judith A. Stafford, Dorina C. Petriu, Jens Happe, and Paola Inverardi, editors, *QoSA/ISARCS*, pages 33–42. ACM, 2011. doi:10.1145/2000259.2000267. (cited on page 21).

[KM06] Ronny Kolb and Dirk Muthig. Architecture-centric quality engineering form software product lines. In *SPLC*, page 226. IEEE Computer Society, 2006. URL: http://dl.acm.org/citation.cfm?id=1158337.1158710. (cited on page 25).

[Koz11] Anne Koziolek. *Automated Improvement of Software Architecture Models for Performance and Other Quality Attributes.* PhD thesis, Karlsruhe Institute of Technology, 2011. http://d-nb.info/1017321884. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000024955. (cited on pages 13 and 14).

[Kru95] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995. doi:10.1109/52.469759. (cited on page 11).

[Kru12] Johannes W. Kruisselbrink. *Evolution Strategies for Robust Optimization.* PhD thesis, Leiden Institute of Advanced Computer Science (LIACS), Faculty of Science, Leiden University, 2012. URL: https://openaccess.leidenuniv.nl/handle/1887/18931. (cited on pages 8 and 9).

[LCL10] Rui Li, Michel R. V. Chaudron, and René C. Ladan. Towards automated software architectures design using model transformations and evolutionary algorithms. In Martin Pelikan and Jürgen Branke, editors, *GECCO (Companion)*, pages 2097–2098. ACM, 2010. doi:10.1145/1830761.1830880. (cited on pages 17 and 27).

[LGHT08] Martin Lukasiewycz, Michael Glaß, Christian Haubelt, and Jürgen Teich. Efficient symbolic multi-objective design space exploration. In *Proceedings of the 13th Asia South Pacific Design Automation Conference, ASP-DAC 2008, Seoul, Korea, January 21-24, 2008*, pages 691–696. IEEE, 2008. doi:10.1109/ASPDAC.2008.4484040. (cited on page 20).

[LGRT11] Martin Lukasiewycz, Michael Glaß, Felix Reimann, and Jürgen Teich. Opt4J: a modular framework for meta-heuristic optimization. In Natalio Krasnogor and Pier Luca Lanzi, editors, *GECCO*, pages 1723–1730. ACM, 2011. doi:10.1145/2001576.2001808. (cited on page 33).

[LH12] Kenneth Lind and Rogardt Heldal. Automotive system development using reference architectures. In Jonathan P. Bowen, Huibiao Zhu, and Mike Hinchey, editors, *SEW*, pages 42–51. IEEE Computer Society, 2012. doi:10.1109/SEW.2012.11. (cited on page 82).

[LSBB03] G. Latif-Shabgahi, S. Bennett, and J.M. Bass. Smoothing voter: a novel voting algorithm for handling multiple errors in fault-tolerant control systems. *Microprocessors and Microsystems*, 27(7):303–313, 2003. URL: http://www.sciencedirect.com/science/article/pii/S0141933103000401, doi:10.1016/S0141-9331(03)00040-1. (cited on pages 88 and 102).

[MAK+10] Anne Martens, Danilo Ardagna, Heiko Koziolek, Raffaela Mirandola, and Ralf Reussner. A Hybrid Approach for Multi-attribute QoS Optimisation in Component Based Software Systems. In Heineman et al. [HKP10], pages 84–101. doi:10.1007/978-3-642-13821-8_8. (cited on page 21).

[MBAG10] Indika Meedeniya, Barbora Buhnova, Aldeida Aleti, and Lars Grunske. Architecture-Driven Reliability and Energy Optimization for Complex Embedded Systems. In Heineman et al. [HKP10], pages 52–67. doi:10.1007/978-3-642-13821-8_6. (cited on page 18).

[Mit98] Melanie Mitchell. *An introduction to genetic algorithms*. MIT Press, 1998. URL: http://dl.acm.org/citation.cfm?id=522098. (cited on page 8).

[MKBR10] Anne Martens, Heiko Koziolek, Steffen Becker, and Ralf Reussner. Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In Alan Adamson, Andre B. Bondi, Carlos Juiz, and Mark S. Squillante, editors, *WOSP/SIPEW*, pages 105–116. ACM, 2010. doi:10.1145/1712605.1712624. (cited on pages 2, 17, 52, 53 and 86).

[NHAH04] Masanori Natsui, Naofumi Homma, Takafumi Aoki, and Tatsuo Higuchi. Topology-oriented design of analog circuits based on evolutionary graph generation. In Xin Yao and et al., editors, *PPSN*, volume 3242 of *Lecture Notes in Computer Science*, pages 342–351. Springer, 2004. doi:10.1007/978-3-540-30217-9_35. (cited on page 100).

[OMGa] OMG (Object Management Group). UML Superstructure Specification, v2.0. URL: http://www.omg.org/cgi-bin/doc?formal/05-07-04. (cited on page 13).

[OMGb] OMG (Object Management Group). UML/MARTE (UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems). URL: http://www.omg.org/spec/MARTE/1.1/. (cited on page 27).

[PB99] Vreda Pieterse and Paul E. Black. *Algorithms and Theory of Computation Handbook*. CRC Press LLC, "Levenshtein distance", in Dictionary of Algorithms and Data Structures [online], 1999. URL: http://www.nist.gov/dads/HTML/Levenshtein.html. (cited on page 119).

[PEP06] Andy D. Pimentel, Cagkan Erbas, and Simon Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. Computers*, 55(2):99–112, 2006. doi:10.1109/TC.2006.16. (cited on page 21).

[PHS$^+$04] Hector Posadas, Fernando Herrera, Pablo Sánchez, Eugenio Villar, and Francisco Blasco. System-level performance analysis in systemc. In *2004 Design, Automation and Test in Europe Conference and Exposition (DATE 2004), 16-20 February 2004, Paris, France*, pages 378–383. IEEE Computer Society, 2004. doi:10.1109/DATE.2004.1268876. (cited on pages 63 and 81).

[Pim08] Andy D. Pimentel. The Artemis workbench for system-level performance evaluation of embedded systems. *International Journal of Embedded Systems*, 3(3):181–196, 2008. doi:10.1504/IJES.2008.020299. (cited on page 19).

[PSZ05] Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. Multi-objective design space exploration of embedded systems. *J. Embedded Computing*, 1(3):305–316, 2005. URL: http://iospress.metapress.com/content/1yd2pya4avy0799x/. (cited on page 18).

[RHRR12] Per Runeson, Martin Höst, Austen Rainer, and Björn Regnell. *Case Study Research in Software Engineering - Guidelines and Examples*. Wiley, 2012. URL: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-1118104358.html. (cited on pages 81 and 131).

[RKM11] Outi Räihä, Kai Koskimies, and Erkki Mäkinen. Multi-objective genetic synthesis of software architecture. In Natalio Krasnogor and Pier Luca Lanzi, editors, *GECCO (Companion)*, pages 249–250. ACM, 2011. doi:10.1145/2001858.2001998. (cited on page 20).

[Rot06] Octavian Paul Rotaru. Caching patterns and implementation. *Leonardo Journal of Sciences*, 8(8):61–76, January-June 2006. URL: http://ljs.academicdirect.org/A08/61_76.htm. (cited on page 100).

[SELC12] Oliver Schütze, Xavier Esquivel, Adriana Lara, and Carlos A. Coello Coello. Using the averaged hausdorff distance as a performance measure in evolutionary multiobjective optimization. *IEEE Trans. Evolutionary Computation*, 16(4):504–522, 2012. doi:10.1109/TEVC.2011.2161872. (cited on page 109).

[SIMA13] Abdel Salam Sayyad, Joseph Ingram, Tim Menzies, and Hany Ammar. Scalable product line configuration: A straw to break the camel's back. In *ASE*, pages 465–474. IEEE, 2013. doi:10.1109/ASE.2013.6693104. (cited on page 24).

[SMA13] Abdel Salam Sayyad, Tim Menzies, and Hany Ammar. On the value of user preferences in search-based software engineering: a case study in software product lines. In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 492–501. IEEE / ACM, 2013. URL: http://dl.acm.org/citation.cfm?id=2486853. (cited on page 24).

[SPG10] Christopher L. Simons, Ian C. Parmee, and Rhys Gwynllyw. Interactive, evolutionary search in upstream object-oriented class design. *IEEE Trans. Software Eng.*, 36(6):798–816, 2010. doi:10.1109/TSE.2010.34. (cited on page 22).

[STT+08] Guido Sand, Jochen Till, Thomas Tometzki, Maren Urselmann, Sebastian Engell, and Michael Emmerich. Engineered versus standard evolutionary algorithms: A case study in batch scheduling with recourse. *Computers & Chemical Engineering*, 32(11):2706–2722, 2008. doi:10.1016/j.compchemeng.2007.09.006. (cited on page 100).

[Sun] Sun Microsystems (merged with Oracle Corporation). Java: a set of several computer software products and specifications from Sun Microsystems (which has since merged with Oracle Corporation), v6. URL: http://java.sun.com/. (cited on pages 43 and 47).

[Szy98] Clemens A. Szyperski. *Component software - beyond object-oriented programming*. Addison-Wesley-Longman, 1998. URL: http://dl.acm.org/citation.cfm?id=515228. (cited on page 12).

[Thea] The Apache™ Hadoop® project. http://hadoop.apache.org/. URL: http://hadoop.apache.org/. (cited on page 136).

[Theb] The Distributed ASCI Supercomputer 4. http://www.cs.vu.nl/das4/. URL: http://www.cs.vu.nl/das4/. (cited on page 139).

[TK11] Catia Trubiani and Anne Koziolek. Detection and solution of software performance antipatterns in palladio architectural models. In Samuel Kounev, Vittorio Cortellessa, Raffaela Mirandola, and David J. Lilja, editors, *ICPE*, pages 19–30. ACM, 2011. doi:10.1145/1958746.1958755. (cited on pages 22 and 99).

[TMD10] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software Architecture - Foundations, Theory, and Practice*. Wiley, 2010. URL: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-EHEP000180.html. (cited on pages 10, 11 and 13).

[Tru11] Catia Trubiani. *Automated generation of architectural feedback from software performance analysis results*. PhD thesis, Universita di L'Aquila, 2011. URL: www.di.univaq.it/catia.trubiani/phDthesis/PhDThesis-CatiaTrubiani.pdf. (cited on page 103).

[vdLSR07] Frank van der Linden, Klaus Schmid, and Eelco Rommes. *Software product lines in action - the best industrial practice in product line engineering*. Springer, 2007. doi:10.1007/978-3-540-71437-8. (cited on pages 2 and 14).

[WTVL06] Ernesto Wandeler, Lothar Thiele, Marcel Verhoef, and Paul Lieverse. System architecture evaluation using modular performance analysis: a case study. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(6):649–667, 2006. doi:10.1007/s10009-006-0019-5. (cited on page 41).

[WW04] Xiuping Wu and C. Murray Woodside. Performance modeling from software components. In Jozo J. Dujmovic, Virgílio A. F. Almeida, and Doug Lea, editors, *WOSP*, pages 290–301. ACM, 2004. doi:10.1145/974044.974089. (cited on page 52).

[YCAM+11] Chantal Ykman-Couvreur, Prabhat Avasare, Giovanni Mariani, Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. Linking run-time resource management of embedded multi-core platforms with automated design-time exploration. *IET Computers & Digital Techniques*, 5(2):123–135, 2011. doi:10.1049/iet-cdt.2010.0030. (cited on page 18).

[YD10] Adel Younis and Zuomin Dong. Trends, features, and tests of common and recently introduced global optimization methods. *Engineering Optimization*, 42(8):691–718, 2010. doi:10.1080/03052150903386674. (cited on pages 31 and 33).

[ZLT02] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. Spea2: Improving the strength pareto evolutionary algorithm for multiobjective optimization. In *Evolutionary Methods for Design, Optimisation, and Control*, pages 95–100. CIMNE, Barcelona, Spain, 2002. URL: http://www.tik.ee.ethz.ch/file/97b63e1661aedd98422e486b7c888ad7/ZLT2002a.pdf. (cited on pages 33 and 52).

[ZT98] Eckart Zitzler and Lothar Thiele. Multiobjective optimization using evolutionary algorithms - a comparative case study. In A. E. Eiben, Thomas Bäck, Marc Schoenauer, and Hans-Paul Schwefel, editors, *PPSN*, volume 1498 of *Lecture Notes in Computer Science*, pages 292–304. Springer, 1998. doi:10.1007/BFb0056872. (cited on page 10).

Summary

Software architecting is a people-intensive, non-trivial and demanding task for software engineers to perform. At the same time, its architecting is a fundamental activity of software development because it involves several questions such as balancing the dependencies among components, maximization modularity, and fulfilling of quality requirements. The architecture is a key enabler for software systems. Besides being crucial for user functionality, the software architecture has deep impact on non-functional properties such as performance, safety, energy consumption and cost. Moreover, software architecture addresses fundamental design choices that are often difficult or very expensive to change in later stages of development. Hence, methods and techniques are needed for designing good software architectures which meet various quality constraints in the the early phases of development.

In this dissertation, an automated approach for software architecture design, named ***AQOSA*** (***A***utomated ***Q***uality-driven ***O***ptimization of ***S***oftware ***A***rchitectures), is proposed that supports analysis and optimization of multiple quality attributes both for single products as well as for product lines:

First of all, we demonstrate of a meta-heuristic optimization approach for automated software architecture design in a real industrial system. More specifically, it reports the results of applying our architecture optimization framework to an automotive sub-system that was conducted based on a large-scale real world industrial case study. The framework supports multiple quality attributes including response time, processor utilization, bus utilization, safety and cost.

Moreover, we introduce two novel degrees of freedom for the optimization of software architectures. It presents the usefulness of the topology degree of freedom as well as replication-degree of freedom. It demonstrates how the number of processing nodes and their interconnecting network can be codified to fit into a genetic algorithm

genotype and thus be subject to automated synthesis. Our studies show that these extra degrees of freedom lead to better overall software architecture optimization. Moreover, it analyses the effectiveness of these two new degrees of freedom by running a very computationally-intensive experiment against our industrial case study. The results of this case study show us additional evidence for the usefulness of these two novel degrees of freedom.

In addition to that, we compare between various combinations of evolutionary algorithm search operators (both domain-specific and generic) for multi-objective optimization of software architecture. The domain-specific operators we study are motivated by software architectural anti-patterns. However, each heuristic-based search operator improves only one quality attribute of the solution, which is challenging for multi-objective problems. To address this issue, we develop strategies for mixing generic and domain-specific search operators in evolutionary algorithms that speed up the finding of good solutions.

Finally, we propose a new search-based approach for generating a set of optimal software architectural solutions for use in software product lines. This approach extends our architecture optimization framework in the direction of features. In our new approach, we add feature models as input to the framework and take into account the relationship between the software components in the architecture and features in the feature model. The AQOSA optimizer addresses this by first searching for architectures that are optimal for individual product. After that, it analyses the commonality of the found optimal solutions and proposes a set of solutions which are suitable for the range of products defined by various feature combinations.

Samenvatting

Het ontwerpen van een software-architectuur is voor software engineers een arbeidsintensieve, niet-geringe en veeleisende taak om uit te voeren. Tegelijkertijd is het ontwerpen van een architectuur een fundamenteel onderdeel van softwareontwikkeling, omdat het verscheidene kwesties aanroert zoals het uitbalanceren van de afhankelijkheden tussen componenten, het maximaliseren van de modulariteit en het voldoen aan kwaliteitseisen. De architectuur vervult een sleutelrol in softwaresystemen. Naast dat software-architectuur cruciaal is voor gebruikersfunctionaliteit, heeft het grote invloed op niet-functionele eisen zoals goede prestaties, veiligheid, energieverbruik en kosten. Bovendien richt software-architectuur zich op fundamentele ontwerpkeuzes die in latere ontwikkelstadia vaak moeilijk of erg kostbaar zijn om te herzien. Vandaar dat er methoden en technieken nodig zijn om in een vroeg ontwikkelstadium software-architecturen te ontwerpen die aan diverse kwaliteitseisen voldoen.

In dit proefschrift wordt een geautomatiseerde aanpak voor het ontwerpen van een software-architectuur voorgelegd, genaamd AQOSA (Automated Quality-driven Optimization of Software Architectures), die het mogelijk maakt om meerdere kwaliteitskenmerken te analyseren en te optimaliseren voor zowel afzonderlijke producten als productlijnen:

Ten eerste tonen we een metaheuristische optimaliseringstechniek voor het geautomatiseerd ontwerpen van een software-architectuur in een industrieel systeem. We doen met name verslag van de resultaten die zijn verkregen door het toepassen van ons architectuuroptimaliseringsraamwerk op een automobiel deelsysteem, gebaseerd op een grootschalige, industriële casestudy. Het raamwerk biedt ondersteuning voor meerdere kwaliteitskenmerken inclusief reactietijd, processorbezetting, busbezetting, veiligheid en kosten.

Bovendien introduceren we twee nieuwe vrijheidsgraden voor het optimaliseren van software-architecturen. Het laat het nut zien van de vrijheidsgraden betreffende topologie en replicatie. Het legt uit hoe het aantal verwerkingsknooppunten en hun onderlinge verbindingen kunnen worden gecodificeerd binnen een fenotype in een genetisch algoritme en zo kunnen worden onderworpen aan automatische synthese. Ons onderzoek wijst uit dat deze extra vrijheidsgraden leiden tot een betere optimalisering van software-architecturen. Bovendien analyseert het de effectiviteit van deze twee nieuwe vrijheidsgraden door het vergelijken van een zeer rekenintensief experiment met onze industriële casestudy. De resultaten van deze casestudy geven ons aanvullende bewijzen voor het nut van deze twee nieuwe vrijheidsgraden.

Daarnaast vergelijken we verschillende combinaties van zoekbewerkingen voor evolutionaire algoritmen (zowel domeinspecifieke als algemene) voor multi-objective optimalisering van software-architectuur. Software-architectonische antipatronen vormen de aanleiding voor de domeinspecifieke bewerkingen die we bestuderen. Echter, elke heuristische zoekbewerking verbetert slechts één kwaliteitskenmerk van de oplossing, hetgeen een uitdaging vormt voor multi-objective problemen. Om dit punt te behandelen ontwikkelen we strategieën voor het combineren van algemene en domeinspecifieke zoekbewerkingen in evolutiealgoritmen die het vinden van goede oplossingen versnellen.

Ten slotte komen we met een nieuwe zoekgebaseerde benadering voor het genereren van een verzameling optimale software-architectonische oplossingen die gebruikt kunnen worden in softwareproductlijnen. Deze benadering breidt ons architectuur-optimaliseringsraamwerk uit met productkenmerken. In onze nieuwe benadering geven we modellen voor productkenmerken als invoer aan het raamwerk en houden we rekening met het verband tussen softwarecomponenten in de architectuur en de productkenmerken in het model. De AQOSA-optimaliseerder doet dit door eerst te zoeken naar architecturen die optimaal zijn voor een individueel product. Daarna analyseert hij de overeenkomsten in de gevonden optimale oplossingen, waarna hij een verzameling oplossingen aanbiedt die geschikt zijn voor de reeks producten die door de verschillende combinaties van productkenmerken zijn gedefinieerd.

List of Publications

The work described in this dissertation has resulted in the following publications:

Journal Articles

[1] Ramin Etemaadi, Kenneth Lind, Rogardt Heldal, and Michel R. V. Chaudron. *Quality-driven optimization of system architecture: Industrial case study on an automotive sub-system.* Journal of Systems and Software, 86(10):2559–2573, 2013.
doi: 10.1016/j.jss.2013.05.109

[2] Ramin Etemaadi and Michel R. V. Chaudron. *New Degrees of Freedom in Meta-heuristic Optimization of Component-Based Systems Architecture: Architecture Topology and Load Balancing.* Science of Computer Programming Journal - Special Issue of Best Papers from Euromicro-SEAA 2012.
doi: 10.1016/j.scico.2014.06.012

Journal Articles [Draft version]

[3] Ramin Etemaadi, Kenneth Lind, and Michel R. V. Chaudron. *Multiobjective Quality-Driven Architecture Optimization for Software Product Lines.* Draft version for submission to Journal of Systems Architecture (JSA).

Peer-Reviewed Conference/Workshop Papers

[4] Ramin Etemaadi and Michel R. V. Chaudron. *Varying Topology of Component-based System Architectures using Metaheuristic Optimization.* In Vittorio Cortellessa, Henry Muccini, and Onur Demirors, editors. 38th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2012, Cesme, Izmir, Turkey, September 5-8, 2012, pages 63–70.
doi: 10.1109/SEAA.2012.38

[5] Ramin Etemaadi and Michel R.V. Chaudron. *Combinations of Antipattern Heuristics in Software Architecture Optimization for Embedded Systems.* In Proceedings of the 6th International Workshop on Model Based Architecting and Construction of Embedded Systems (ACES-MB 2013), co-located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2013), Miami, Florida, USA, September 29th, 2013, volume 1084 of CEUR Workshop Proceedings. CEUR-WS.org, 2013.
CEUR link: http://ceur-ws.org/Vol-1084/

[6] Ramin Etemaadi and Michel R.V. Chaudron. *Distributed Optimization on Super Computers: Case Study on Software Architecture Optimization Framework.* In Proceedings of the 3rd International Workshop on Evolutionary Computation Software Systems (EvoSoft 2014), GECCO Comp 2014 - the 2014 Conference Companion on Genetic and Evolutionary Computation Companion, Vancouver, Canada, July 12th, 2014, pages 1125-1132.
doi: 10.1145/2598394.2605686

[7] Rui Li, Ramin Etemaadi, Michael T. M. Emmerich, and Michel R. V. Chaudron. *An Evolutionary Multiobjective Optimization Approach to Component-Based Software Architecture Design.* In Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2011, New Orleans, LA, USA, 5-8 June 2011, pages 432–439.
doi: 10.1109/CEC.2011.5949650

[8] Ramin Etemaadi, Michael T. M. Emmerich, and Michel R. V. Chaudron. *Problem-specific Search Operators for Metaheuristic Software Architecture Design.* In Gordon Fraser and Jerffeson Teixeira de Souza, editors. Search Based Software Engineering - 4th International Symposium, SSBSE 2012, Riva del Garda, Italy, September 28-30, 2012. Proceedings, volume 7515 of Lecture Notes in Computer Science. Springer, 2012, pages 267–272.
doi: 10.1007/978-3-642-33119-0_20

[9] Ramin Etamaadi and Michel R.V. Chaudron. *A Model-based Tool for Automated Quality-driven Design of System Architectures.* In Joint Proceeedings of co-located events as the ECMFA 2012: 8th European Conference on Modelling Foundations and Applications, 2012, Lyngby, Denmark, July 2-5, 2012, pages 324–327.

Acknowledgements

You are holding a book that results from four years of research work. During this path, I owe thanks to many generous and supportive people. It is a great pleasure for me to thank them for supporting me during my Ph.D. studies.

First and foremost, I would like to thank prof. dr. Michel R. V. Chaudron, who accepted me as a Ph.D. student and opened up the gates to the scientific world. I enjoyed his positive attitude to make a friendly working environment.

I really appreciated the opportunity to collaborate with LIACS institute, I have to thank prof. dr. Thomas H. W. Bäck and prof. dr. Joost N. Kok for their support.

After that, I also would like to send great thanks to dr. Rui Li for his kindness and helpfulness. Because of his great support I have been able to start this journey. I also would like to express my sincere gratitude to dr. Michael T. M. Emmerich for his welcoming attitude toward my questions.

My very special thanks go to dr. Kenneth Lind and dr. Rogardt Heldal for their wonderful contribution to this study. I am greatly honoured and feel very fortunate to meet you during this study.

I am also grateful to my colleagues and friends for being a constant source of inspiration during this Ph.D. period. Especially, I am thankful to Ana, Ariadi, Behrooz, Bilal, Dave, Edgar, Hafeez, Hossein, Javier, Johannes, Meghdad, Sahar, and Werner.

Mom and dad, my deepest gratitude goes to you. Thank you for your love and support, and for encouraging me to pursue my education.

Most importantly, I would like to thank the love of my life, my wonderful wife, **NAFISEH** for her endless support and understanding. Without her love, I would not have been able to overcome the challenges during this period, and this dissertation would not have been possible without her patience, and continuous support.

About the Author

Ramin Etemaadi was born in 1981 in Tehran, Iran. He graduated his B.Sc. with the honours degree in computer science from Isfahan University of Technology, Iran, in 2002. He received his M.Sc. in 2005, and the master thesis was entitled “ Software Creation: Automatic Design for Object-Oriented Software ”. From September 2010, he worked as a Ph.D. candidate at the Leiden Institute of Advanced Computer Science (LIACS) at the Leiden University Faculty of Science. He worked within the Software Engineering Group under supervision of Prof. Dr. Michel R. V. Chaudron. His research interests include software architecture, software design, component-based software engineering, and automation of software development process.

Since 2002, Ramin has been involved as a software architect and technology lead in various software development projects. He participated in the development of the first e-commerce solution of Iran (named Pardakht), which won “ 2nd National SheikhBahai Entrepreneurship Festival ”. As the managing director of a business unit, he leaded a small start-up team to become a very successful business. The service became largest mobile added-value-service in Iran, and the major service provider for the mobile operators in the country.

Currently, he is working as a Senior Research Engineer in Exact research department in Delft, Netherlands. Exact is a leading global supplier of complete ERP solutions for small and medium-sized enterprises. Exact develops award-winning business software for companies around the globe and supports over 100,000 small to medium-sized enterprises in more than 125 countries with their daily management.