UNIVERSITY OF GOTHENBURG

Generating System Architectures from a Given Feature Model

Feature-based multi-objective optimization

Master of Science Thesis in the Programme Computer Science

VASILEIOS KARAGIORGIS

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, January 2014

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.
The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Generating System Architectures from a Given Feature Model
Feature-based multi-objective optimization

VASILEIOS KARAGIORGIS

© VASILEIOS KARAGIORGIS, January 2014.

Examiner: Michel Chaudron

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

Department of Computer Science and Engineering
Göteborg, Sweden January 2014

Abstract

Until now, software architects could create an architecture for a future system and analyze its software and hardware components to see if it meets the quality requirements. They also can use automated tools to optimize the initial design and find the best alternative architectures which meet the quality requirements. If the system is also part of a family of similar systems (software product line), then this means that it implements some features of this family. Moreover, it means that any additional similar systems could reuse some components from the initial system. However, if there is a need for other similar systems (products), then the software architect would have to design different architectures to meet the different quality requirements of those products. The architect would not be able to reuse the old architecture to create the new one and they would also need to optimize the architecture of every product. The existing optimization tools are unable to optimize one architecture to cover many products.

This work proposes a new automated method, which optimizes a software architecture based on the feature model it belongs and the selected products. In this approach, the optimization runs only for the architectures that support the defined products and then it performs commonality analysis to find the optimal architectures that can support all the selected products.

The method is tested on an industrial case study with 480 different possible products, and found many sufficiently similar common solutions, which can support all of the selected products.

Keywords:
Software Product Line (SPL), Multi-objective Optimization, Genetic Algorithms, Feature Models, Software Architecture

Acknowledgements

I, particularly, would like to thank my supervising professor, Michel Chaudron, for all his personal efforts to help me complete this thesis in a tight schedule, despite the difficulties that were emerged.

Moreover, I want to thank Ramin Etemaadi for his excellent collaboration we had, on extending the AQOSA framework to fit the purposes of this thesis.

Last but not least, this thesis would not be completed without the understanding and caring support of my life companion, Sevi, in both good and bad times. I dedicate this work to her.

Table of contents

Abstract 3
Acknowledgements 4
Table of contents 5
List of Figures 7
List of tables 7
Chapter 1: Introduction 8
1.1 Introduction 8
1.2 Problem Description 8
1.3 Contribution 9
1.4 Outline 9
Chapter 2: Related Work 10
2.1 Optimization Tools 10
2.1.1 AQOSA 10
2.1.2 ArcheOpterix 10
2.1.3 PerOpteryx 11
2.2 Feature Model Optimization Tools 11
2.3 Feature Modeling Tools 11
2.3.1 S2T2 Configurator 11
2.3.2 Software Product Lines Online Tools (SPLOT) 12
2.3.3 Palladio Feature Model 12
2.3.4 EMF Feature Model 12
2.3.5 Clafer Tools 13
2.3.6 Alloy 13
Chapter 3: Features 14
3.1 Definition 14
3.2 Software Product Lines 14
3.3 Feature Model 14
3.4 Feature configurations 15
3.5 Conclusion 16
Chapter 4: Genetic Algorithms 17
4.1 How they work 17
4.2 Genetic Operators 18
4.2.1 Crossover 18
4.2.2 Mutate 18
4.3 Conclusion 19

Chapter 5: AQOSA framework 20
5.1 Overview 20
5.2 Process 20
5.3 Architecture Modeling 21
5.4 Architecture Optimization 21
5.5 Output 23
5.6 Conclusion 23
Chapter 6: Methodology 24
6.1 New process 24
6.2 Feature modeling tools 26
6.3 How to link features 27
6.4 Optimization for multiple products 29
6.5 Commonality analysis of solutions 29
6.5.1 Solutions distance algorithm (DA) 29
6.5.2 Sum of minimum distances algorithm (SMDA) 31
6.5.3 Common solutions within given delta (GDA) 33
6.5.4 Weaknesses 34
6.6 Conclusion 35
Chapter 7: Case Study 36
7.1 Forming the case study 36
7.2 Products 41
7.2.1 Car1 42
7.2.2 Car2 43
7.2.3 Car3 44
7.2.4 Car4 45
7.2.5 Car5 46
7.3 Experiment 46
7.4 Results 47
7.4.1 Interpretation of results 48
7.5 Limitations 51
7.4 Conclusion 52
Chapter 8: Conclusion 53
8.1 Summary 53
8.2 Future Work 53
References 55

List of Figures

1. Example of using Clafer language to represent a feature model (Clafer: Lightweight Modeling Language, 2013) ... 13
2. Feature Model with selected features (Segura, 2009) ... 15
3. a) Chromosome inheritance with crossover operator (One Point Crossover, 2013), b) Same example in bit string format (Ai-junkie, n.d.) ... 18
4. Overview of AQOSA architecture (Etemaadi, et al., 2013) ... 20
5. Genotype of an architecture in AQOSA (Etemaadi, et al., 2013) ... 22
6. Example of a Pareto front of optimized solutions for Cost and Response time ... 23
7. The new proposed process of AQOSA ... 25
8. Representation, in Clafer, of the feature model used in the case study ... 26
9. The new AQOSA meta-model showing the addition of features and their relations ... 28
10. Pseudo code of distance algorithm ... 30
11. Pseudo code of SMDA ... 32
12. Pseudo code of GDA ... 33
13. The feature model of the case study ... 37
14. Sequence diagram of FullyAdaptiveCruiseControl system operation ... 40
15. Frequency graph of configurations classified by the bandwidth needs of their related components ... 41
16. Frequency graph of configurations classified by the processing needs of their related components ... 42
17. Selected features of Car 1 ... 42
18. Selected features of Car 2 ... 43
19. Selected features of Car 3 ... 44
20. Selected features of Car 4 ... 45
21. Selected features of Car 5 ... 46
22. Explanation of an optimized architecture (solution) as displayed by AQOSA ... 48
23. An extract from the results of GDA showing the 3 closest common solutions ... 49
24. An extract from the results of SMDA, showing the total distance of the common solutions in a sorted list, and the 3 best common solutions below ... 49
25. A string output of a common solution and its related ones, found by GDA ... 49
26. Graphical display of the architecture of the common solution (Car1) in Figure 23. Note the 3 processors of 100, 600 and 100 MHz ... 50
27. Architectures of a) Car2, b) Car3, c) Car4, d) Car5 ... 50
28. Common solutions connected to their related solutions. The lines show the distance between them ... 51

List of tables

1. Example showing the steps of the distance algorithm working on two lists of processors . 31
2. Relation of features with the software components ... 39
3. Number of common solutions found by GDA in various delta values ... 47
4. Comparative results of the total distance of the best common solutions ... 48

Chapter 1: Introduction

1.1 Introduction

Component-based architecture is an architectural approach where the system to be developed is considered a set of hardware and software components linked together. Thus it is possible to measure and analyze certain aspects of the system, such as its quality properties (cost, safety, availability, etc.), before any implementation takes place, allowing the architect to revise the original plan and to find another architecture that fulfills the system requirements. (Li, et al., 2011)

Finding and generating alternative architectures is a complex optimization problem, called single objective or multi-objective optimization problem depending on whether the architecture has to be optimized for only one property or many. The research community addresses this problem using various methods, such as genetic / evolutionary algorithms. These algorithms mimic the natural process of evolution according to which the fittest genes are passed to the next generation. Tools have been developed for single objective optimization (Grunske, et al. 2007) and tools, such as the AQOSA framework (Etemaadi, et al., 2013), for multi-objective optimization.

However, a system can also be part of a series of systems with common characteristics or features. The set of features that the system should include usually is modeled in a tree-like diagram and called “feature model” (Lee, Kang and Lee, 2002). None of the component-based architectures or tools considers the feature model in order to find and generate the alternative architectures. This thesis work aims to bridge this gap by extending the AQOSA framework to be able to link the feature model with the architectural design. This functionality will enable AQOSA to find the optimal design based on the feature model and to find the solutions that are common for many feature configurations.

1.2 Problem Description

As described in introduction, most of the optimization and architecture analysis tools use the component-based architecture as input and consider it as a set of software and hardware components and a relation between them. This enables them to analyze or optimize an architecture based only on software and hardware, for only a specific product. The few tools that consider the notion of feature are either exclusive for feature analysis or use mathematical formulas and assumptions that are problem specific.

However, none of the existing tools consider the notion of system features as components combined with the software and hardware. Even though it is possible to find the most optimal architecture for a system, it is still unknown if this architecture is suitable for the selected feature model and if it can support multiple feature configurations.

The research question that this work is trying to answer is: “Given a feature model, which architecture supports the most features?” This study will use the problem solving case-study

research approach to answer the research question, because it is focused on a practical problem (Yin, 1994)

1.3 Contribution

The purpose of this study is to provide an extension to the AQOSA framework, which will enable the generation of alternative architectures based on a given feature model. This would allow someone, not only find the most suitable architecture for a selected configuration, but also to find which architecture can support the most features or similar top-bottom analysis.

This extension aims to bridge the gap between the software product line community and the software architecture community, in the sense that the architecture community currently benefits from the existing optimization approaches but not when they apply software product line methods for their systems.

It is expected to be used by practitioners that use the tool to find optimized architectures and would like to also add software product line nature in their architecture. Moreover, the mapping between an architecture and a feature model provides a starting ground for researchers who would like to research on multi-objective optimization techniques for product lines.

1.4 Outline

This thesis is divided into eight chapters. Each chapter builds upon the previous ones to develop a knowledge basis, necessary for interpreting the results in Chapter 7. The remaining chapters are as follows:

- Chapter 2 describes the related work and the tools that were tested.
- Chapter 3 describes the notion of features and feature-based software engineering, which is the core addition of this work.
- Chapter 4 explains how the genetic algorithms work and how they find alternative solutions. They are used by AQOSA to find alternative optimized architectures.
- Chapter 5 presents an overview of the AQOSA framework and how it is used.
- Chapter 6 describes the extension of AQOSA by this work: what tools, processes and algorithms were used, along with a discussion of their weaknesses.
- Chapter 7 shows what case study was used to test the proposed method and how the experiment was set up. It also presents the results with an interpretation of them.
- Chapter 8 is the conclusion of this work. It contains a summary of this thesis and concludes with possible future work.

Chapter 2: Related Work

2.1 Optimization Tools

2.1.1 AQOSA

This work is based on the AQOSA framework initially proposed by Li, et al. (2011) and evolved by Etemaadi, et al. (2013). As mentioned before, the problem at hand was that while component-based architecture designs enable the analysis of quality properties, which may be in conflict with each other, finding alternative architectures that fulfill all the properties simultaneously was done manually, for small-scale systems only.

Their contribution was the introduction of a tool which automated this procedure of finding alternative solutions in respect to all quality attributes and performed analysis, too. The tool considers four quality attributes for every architecture: cost, safety, utilization, and performace, though caters for more. It uses 3 different evolutionary optimization algorithms to find optimized solutions and produces a Pareto front of these solutions to choose from. It was initially tested on a small-scale example but Etemaadi, et al. (2013) applied AQOSA on a real world example, a case study by SAAB. The results show that AQOSA is able to find better architectures, for all quality attributes, than the initial one. They also provide an extensive overview of the tool.

These two papers were the primary papers this study relied on, because they explain everything about the tool, which is the only that performs multi-objective optimization for many quality attributes. They also describe the context in which it can be used.

2.1.2 ArcheOpterix

ArcheOpterix (Aleti, et al., 2009) is a tool very similar to AQOSA. It tries to address the problem of finding the best alternative software architecture (esp. in the embedded systems domain) among conflicting quality requirements.

They introduce a platform to implement different architecture evaluation and optimization algorithms to mitigate that problem. It is an Eclipse-plugin and works with the AADL language. The tool uses genetic algorithms to optimize the architecture and uses two methods to rank the solutions: mapping all the objectives into a single objective function and finding the Pareto front line of non-dominated solutions. The tool evaluates the constraints (memory, localization, co-localization) and quality attributes (data transmission reliability and communication overhead) of an architecture.

The experiment results show that the tool produces sufficient evidence that it can find better quality architectures. The tool is very similar to AQOSA and it addresses the same problem but it only uses two quality attributes for optimization which may make the optimization

problem easier. AQOSA uses four quality attributes which makes the problem more realistic though more difficult, too.

2.1.3 PerOpteryx

PerOpteryx (Koziolek, Koziolek and Reussner, 2010) is another tool similar to AQOSA. They address the same problem of finding alternative architectures based on many quality attributes. They propose a hybrid tool of Rule-based and Metaheuristic approach. In particular, they make use of predefined architectural tactics and multi-objective evolutionary optimization to find and optimize the solutions. These tactics run in the reproduction step of the genetic algorithm to modify a solution towards a better candidate solution. From their results, it seems that the use of these tactics, in the searching the solution space, has reduced optimization runtime by at least 50%. The hybrid genetic algorithm also managed to find meaningful quality architectures to choose from.

AQOSA differs due to the lack of these tactics but could definitely benefit from that. In addition, these domain specific tactics could potentially hinder the generalization of the tool.

2.2 Feature Model Optimization Tools

None of the aforementioned tools consider the features of the system in their optimization process. There are tools that evaluate and optimize feature models but none of them consider the architecture level at the same time. The most related work is that of Etxeberria and Sagardui (2008), where they measure the quality of a software product line architecture by introducing quality properties as features in the architecture. They use mathematical formulas for quantification and evaluation of the product line quality. However, the similarities with this work are few. They evaluate only the software product line architecture (set of features) and they don't consider the system architecture at all. Through mathematical formulas, they map the quality features to other features, while this study tries to map features to components.

2.3 Feature Modeling Tools

The first step of this work was to find a tool in order to model the feature model for the case study and also to be able to find all the different configurations of this feature model. So, there was a need for a feature modeler and a configurator with parsing capabilities, to handle the features with code. After some research, the tools that were tested were: S2T2 Configurator, Software Product Lines Online Tool, Palladio Feature Model, EMF Feature Model, Clafer Tools and Alloy Analyzer. The following sub-sections provide a brief description of every considered tool.

2.3.1 S2T2 Configurator

This is a plug-in for the Eclipse IDE made by the LERO group, the Irish Software Engineering Research Center (Botterweck, Janota and Schneeweiss, 2009), (Botterweck and

Pleuss, 2012). It is possible to create feature models through the visual editor and select configurations manually. The tool has a formal reasoning engine running in the background so one can know due to which constraints some features can or cannot be selected from the feature model. This engine produces a formal representation of the feature model in order to be used by SAT solvers to find the different possible configurations. Although it is very easy to install and to use the graphical editor, it has many bugs and it is very time consuming to create a simple model, because it needs many XMI elements to represent a few elements in the editor. Moreover, the lack of documentation made it even more unappealing and eventually was ruled out.

2.3.2 Software Product Lines Online Tools (SPLOT)

The Software Product Lines Online Tools, or SPLOT, is a family of tools from the University of Waterloo, Canada. (Mendonca, Branco, and Cowan, 2009). It allows somebody to create and save their feature models online. It is easy and very fast to use, with good documentation. It can export the feature model in a *.sxfm* file, which stands for (Simple XML Feature Model) and they provide a Java parser for it. One minor drawback is that the cross-tree constraints, such as mutual exclusiveness, have to be implemented in CNF logical clauses, which are not very intuitive. This tool came close second, because the SXFM parser provides neither a configurator nor an appropriate output for a solver.

2.3.3 Palladio Feature Model

Palladio is a program that can predicts quality of software properties from software architecture models. It comes as a plug-in for Eclipse with a graphical editor and it is made by the Karlsruhe Institute of Technology (2012). The program exports to XMI files that can be easily read by existing parsers and it is easy to create a feature model. Even though it is easier to use than S2T2, it is less expressive making it more difficult to develop complex feature models. The major disadvantages are the lack of good documentation and that the graphical editor seems to be broken.

2.3.4 EMF Feature Model

This is another Eclipse tool (EMF Feature Model, 2013) that it is supposed to help with feature modeling. Instead, it is hard to find it and it seems that it is an incubation project / alpha version. There is no documentation, except for a mailing list, and it was impossible to create any model at all with it.

2.3.5 Clafer Tools

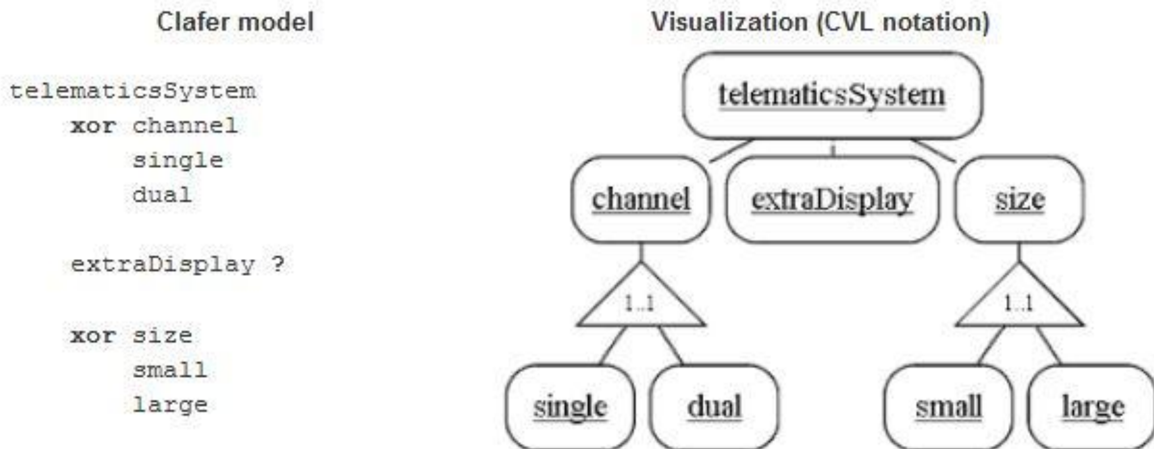

Figure 1: Example of using Clafer language to represent a feature model (Clafer: Lightweight Modeling Language, 2013)

Clafer (Antkiewicz, et. al, 2013) is a lightweight language developed at GSD lab at University of Waterloo and MODELS group at IT university of Copenhagen. Clafer stands for **C**lass - **F**eature – **R**eference, it has simple syntax and it is used for creating feature models, class diagrams, meta-models and similar diagrams with complex constraints (Figure 1). Clafer tools are a group of tools which provide more functionality, such as the compiler, instance generator, configurator, multi-objective optimizer and visualizer. Unfortunately, the only tool that was able to run was the compiler, which can compile the Clafer instructions to instructions for the Alloy tool.

2.3.6 Alloy

Alloy, like Clafer, is a language for describing structures as a collection of constraints (Alloy, 2012). Alloy syntax is not as easy and straightforward as Clafer syntax, but it comes with a tool, Alloy Analyzer. This tool is a solver that tries to find the structures that satisfy the constraints of the original structure. For example, the solver can find all configurations of a feature model and display each individual configuration. This tool comes in a jar file, so that it can be used within a Java program.

Chapter 3: Features

3.1 Definition

The notion of features in software systems was initially introduced by Kang, et al., at the Software Engineering Institute of Carnegie Mellon University, in an effort to create a new analysis method for identifying features in a domain. A feature is "a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or system" (Kang, et al., 1990, p.3). Therefore, a software system can have many features that describe it.

3.2 Software Product Lines

Features are the core of Software Product Lines (SPL). According to the Software Engineering Institute, a SPL is a software engineering paradigm and defined as "a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" (SEI, 2013). In other words, a SPL is a family of related programs, all defined by their set of features they implement.

The idea behind SPL is to build similar software systems that share certain features from a single set of features, called *product line*. It is inspired by the techniques the big manufacturers in the automotive and airspace industry used for mass production. These techniques involved the reuse of parts in different products in order to reduce cost and production time. SPL apply these techniques on software production, by reusing code and other assets to build similar software systems. Unlike the big manufacturers however, in SPL the assets can even come from other external software or markets, such as the so-called *commercial-off-the-shelf* product (COTS products). (Clements and Northrop, 2001)

In the past, software development hoped for opportunistic code reuse, meaning that general code would be used in hope that future opportunities for reuse will arise. SPL allow for targeted decisions to be made, on when and which asset will be created and reused. This leads, in the long run, to quicker development time, easier maintenance and easier transition from one product to the other.

3.3 Feature Model

In SPL, features may have certain variations, constraints and dependencies. All these are included in a model, called *feature model*. Feature models are usually displayed with a hierarchical tree-like diagram, as seen in Figure 2, starting from a root feature which is usually the kind of products the SPL produces. (Lee, Kang and Lee, 2002)

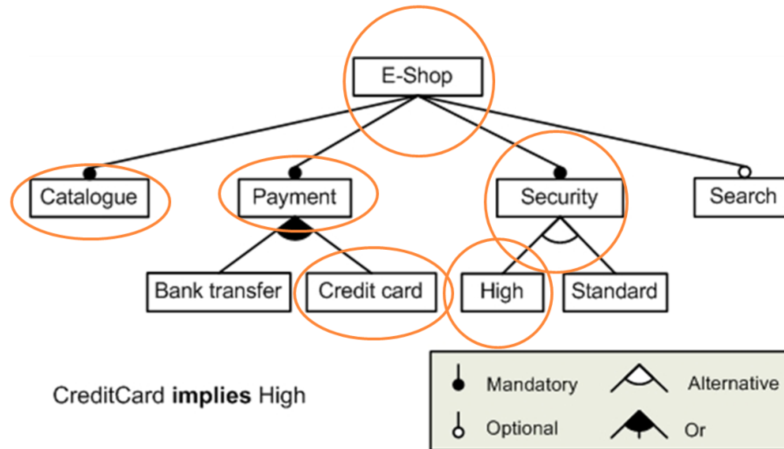

Figure 2: Feature Model with selected features (Segura, 2009)

Features have parent-child relationships with each other, which are displayed as connecting lines. A candidate product can select features from this model but it should be done according to their constraints. Features can be mandatory or optional, marked by black or white circles above their name, respectively. Mandatory features mean that a product must always include these features if their parent feature is selected, while optional features can be left unselected. Another constraint can be which features can be selected from a parent feature. In this case there is the OR selection and ALTERNATIVE (XOR) selection. The OR selection allows the selection of any of the sub-features, even all of them. The XOR selection allows the selection of only one of the sub-features, implying the selection of one mutually excludes the others.

Moreover, there can be constraints between features that cannot be modeled as a parent-child relationship. These are called *cross-tree constraints*, because they relate two sibling features across the tree model. The cross-tree constraints can be any logical clause between two features but they are usually used for *implies* and *excludes* relationships. *Implies* is a relation between two features where one feature requires the other in order to be implemented and therefore if the one is selected then the other must also be selected. The *excludes* relationship means that the selection of one feature excludes the use of the other and therefore cannot be selected together, in the same product.

3.4 Feature configurations

As mentioned before, a product must implement some features from its SPL. A product can select any number of features from the feature model, provided they satisfy the constraints. The set of features that are selected for a product is called *feature configuration* and therefore, a different feature configuration implies a different product. The number of different feature configurations that can be selected (different products), can grow very big because of the feature combinations. In automotive industry, they can have minimum 500 different configurations.

Figure 2 showed an example feature model from an e-shop SPL. This model contains mandatory features (*catalogue*, *payment*, *security*) and one optional feature (*search*). An e-

shop can have any of these payment methods: *bank transfer* and/or *credit card* payment. It can also have either *standard* or *high* security, but not both. However, if the *credit card* payment method is used then it must come with *high security*. This is a cross-tree constraint. The circled features mark a potential feature configuration, where the mandatory features must be selected and the selection of credit card must come with high security.

3.5 Conclusion

This chapter described what a feature is. A feature is a characteristic of a software system and a system can have many features that describe it. Next, it described the paradigm of software product lines in which a system can be part of a series of similar systems and they are all defined by their set of features they implement. The idea is to massively reuse assets to build similar systems.

The features of a product line are usually displayed in a tree-like diagram with parent-child relations and selection constraints. Every product from this product line must select which features it will implement, as long as they satisfy their constraints. This selection is called feature configuration and it defines the product.

Chapter 4: Genetic Algorithms

4.1 How they work

Genetic algorithms are algorithms that mimic the process of evolution, in order to solve search and optimization problems. Optimization is the problem of finding the optimal element from a set of feasible elements. In other words, it is about finding the maximum or minimum of a function by choosing various values and satisfying its constraints (INFORMS Computing Society, 2010). The evolution works on genes of species using various mechanisms, such as reproduction, mutation, combination and selection (survival of the fittest). These algorithms use these mechanisms to evolve a solution until it satisfies the conditions of the problem. (Goldberg, 1989)

In nature, every living being or individual has some set of characteristics, which is called phenotype and distinguishes it from other individuals. The phenotype is specified by the individual's set of chromosomes, the genotype. The genotype tells, by the order of the chromosomes, which characteristics an individual should have. Every individual inherits some chromosomes from one parent and some from the other and can also pass its genes to the next generation through its offspring.

In genetic programming, an optimization problem might have several potential solutions, which are called individuals or phenotypes. Each individual has a set of properties, which are called genotype or chromosomes and are usually represented as a series of binary zeros and ones though other representations are also possible (Whitley, 1994, p.66). The genotype is a combination or mutation of chromosomes from the two parents of the individual. The genetic algorithm starts by generating some random individuals and forms an initial population. This population consists the initial generation, too. The algorithm evaluates every individual with a score using a fitness function and this score is usually the value of the function to be maximized / minimized. Then it selects some of the fittest individuals to generate offsprings for the next generation / iteration, applying the evolution mechanisms (crossover, mutate) to parents. The offsprings form the next generation and the steps start again until the termination criteria are satisfied, such as a maximum number of iterations (Prebys, 2007). The number of the children two parents can produce can vary but it is usually two.

Even though it is difficult to measure the complexity of the genetic algorithms, it is perceived that the computational complexity of the problems they have to solve is related to the search space of the solutions (Rylander and Foster, 2001). Therefore, they converge slower to a solution when the problem is big or complex and need extra computational power to keep their execution time as low as possible.

4.2 Genetic Operators

Genetic operators are operators that are used in order to alter, transfer or combine genetic code to other individuals. Two operators are going to be explained because they are the ones mostly used.

4.2.1 Crossover

Crossover (Bajpai and Kumar, 2010) is an operator for selecting which chromosomes from the parents are going to be transferred to the offspring. There are several methods to do this but the simplest is the one-point crossover. In this method, a single crossover point is set on the chromosome series of both parents. The children should inherit the chromosomes (or bits) from the first parent up until the crossover point and the rest from the second parent, after its crossover point. Figures 3a and 3b illustrate an example.

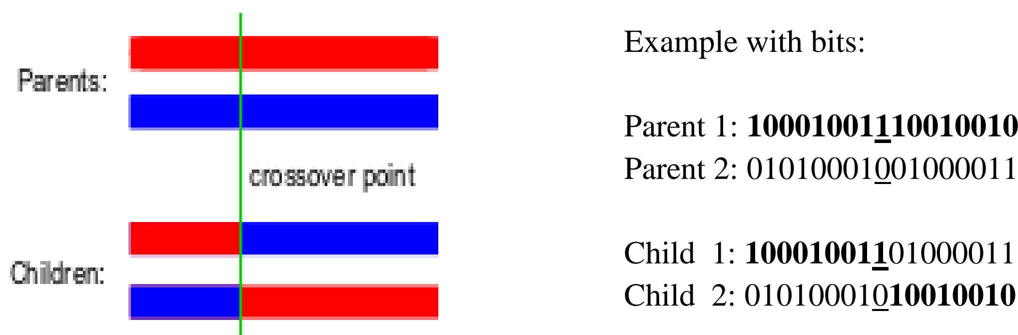

Figure 3: a) Chromosome inheritance with crossover operator (One Point Crossover, 2013), b) Same example in bit string format (Ai-junkie, n.d.)

4.2.2 Mutate

The mutation operator (Bajpai and Kumar, 2010) flips one chromosome from its initial value to something else. The reason is to ensure the diversity of the population; otherwise the solutions would become too similar to each other. There various methods for this operator as well, but only the bit string mutation is going to be explained since it is the one implemented in this work. Bit string mutation is to flip a bit to its opposite state, for example from 0 to 1 or from 1 to 0. The number of mutations in every bit string is defined by a probability value, which is usually quite low. The algorithm performs first the crossover function and then for every bit in the string it calculates, based on the chance, whether a mutation operation should be applied. A mutation example is illustrated below (Wikipedia, 2013):

Example:
1 0 1 0 0 1 0
↓
1 0 1 0 1 1 0

The probability of mutation here is only 1 per string or (1 / string length).

4.3 Conclusion

This chapter described what genetic algorithms are and how they work. They are algorithms that encode a solution of a problem into a bit string and then with various operators (copy, mutate, crossover) they pass certain bits to a child solution. The best solutions towards a goal are passed to the next generation which repeats the same steps. The operators were described individually at the end of this chapter.

Chapter 5: AQOSA framework

5.1 Overview

This work is based on the AQOSA framework by Etemaadi, et al. (2013), which stands for Automated Quality-driven Optimization of Software Architecture. The purpose of this tool is to find architecture designs automatically. The research community converges to two kinds of solutions to achieve this: Rule-based solutions and Metaheuristic-based solutions. Rule-based solutions rely on standard rules and tactics in order to fix and certain ill-performing parts of the architecture. Metaheuristic-based solutions (Blum and Roli, 2003 cited in Etemaadi, et al., 2013, p.2560) try to optimize architecture towards its quality goals, in subsequent iterations, until a good enough solution is found. Every iteration can generate a lot of potential solutions to be optimized in the next iteration, which is called the design space. In order to effectively search this space there are various techniques, such as: genetic algorithms, neural networks and more. AQOSA is a metaheuristic-based tool for many quality objectives. Its search method is genetic algorithms that optimize the software architectures.

5.2 Process

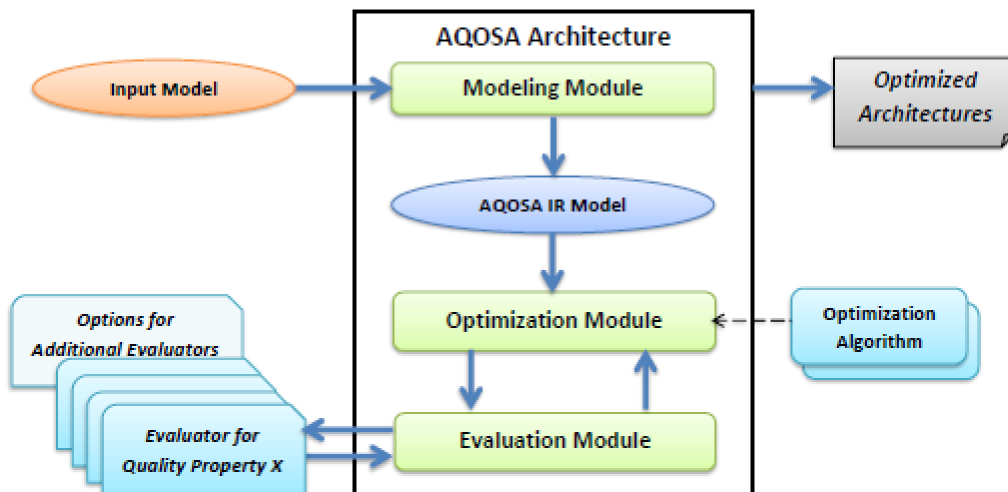

Figure 4: Overview of AQOSA architecture (Etemaadi, et al., 2013)

AQOSA is written in Java and it is built based on a modular architecture as seen in Figure 4. It uses an internal representation (IR) for describing the architectural problem, which is the basis for its core and for the modeling editor. AQOSA works on an input file which contains: 1) the components of the architecture (software, hardware), 2) usage scenarios, 3) the objective function, indicating which quality properties should be considered, 4) the components' specifications. The user can model their input with the model editor and then the AQOSA follows these steps:

1. Create new potential architectures / solutions using genetic algorithms.
2. Evaluate the new solutions based on the selected quality properties.
3. Choose some Pareto-optimal solutions.
4. Repeat step 1 until the quality objectives or the maximum number of generations has been reached.

5.3 Architecture Modeling

In the beginning, the user has to create the input file by modeling the architecture and the problem in the visual editor. What the user can model is defined by the AQOSA-IR. The IR is simply holder classes that hold the elements of the model after parsing and it is defined by a meta-model. The meta-model can be seen later in Figure 9, in section 6.3. The meta-model contains, among others, the four parts required for the input file.

- Assembly: It contains the software components, their services, their flows for interacting with other components and the actions for these interactions.
- Repository: It contains the hardware components (processors and buses) available for usage in the architecture. It also contains the implementation of the software components and their assignment on the hardware components. The optimization phase actually starts from this place. Moreover, it stores various specifications of the software and hardware (i.e. CPU cycles, bus bandwidth, etc.).
- Scenarios: They are the different scenarios the system might run on. They contain time constraints and deadlines.
- Objectives: The quality objectives to be used for optimization.

5.4 Architecture Optimization

The optimization process can be done with various genetic algorithms. The case study of this work is based on the NSGA-II algorithm (Deb, et al., 2002 cited in Etemaadi, et al., 2013, p.2562) (non-dominated sorting based multi-objective evolutionary algorithm), which is implemented based on the Opt4J optimization framework (Lukasiewycz, et al., 2011).

The genotype, used for optimization, includes: the software components, the hardware nodes, the allocation of the former on the latter, the communication (bus) lines and the connection of hardware nodes with these lines. The genotype can be seen in Figure 5.

During the optimization, many alternative architectures are produced using the *copy*, *mutate* and *crossover* genetic operators. However, it can happen that an alternative architecture consists of invalid topology or allocation, for example nodes with no bus lines etc. Therefore, AQOSA filters out these invalid architectures.

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

(b) Nodes Genome

l (No. of Lines)	Line$_1$	Line$_2$	Line$_3$	...	Line$_l$
	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

Figure 5: Genotype of an architecture in AQOSA (Etemaadi, et al., 2013)

5.5 Output

The tool optimizes the architectures with regard to 5 quality attributes: 1) Cost, 2) Processor utilization, 3) Communication line utilization, 4) Response time, 5) Safety. However, the framework's open architecture allows the addition of more custom quality attributes. After the optimization, the tool produces diagrams which show the alternative architectures in respect to pairs of quality attributes, the *Pareto fronts*. Figure 6 shows an example Pareto front of solutions in respect to Response time and Cost.

5.6 Conclusion

This chapter provided an overview of the AQOSA framework, which this work aims to extend. AQOSA uses genetic algorithms to iteratively optimize an initial architecture towards multiple quality objectives. A user can model the architecture (software and hardware components), as well as its specifications, usage scenarios and the quality objectives. The optimization works by changing the allocation of software components on hardware components and the connection of communication lines to hardware components. At the end, the tool displays the resulting architectures in diagrams, such as the Pareto front for two objectives (Figure 6).

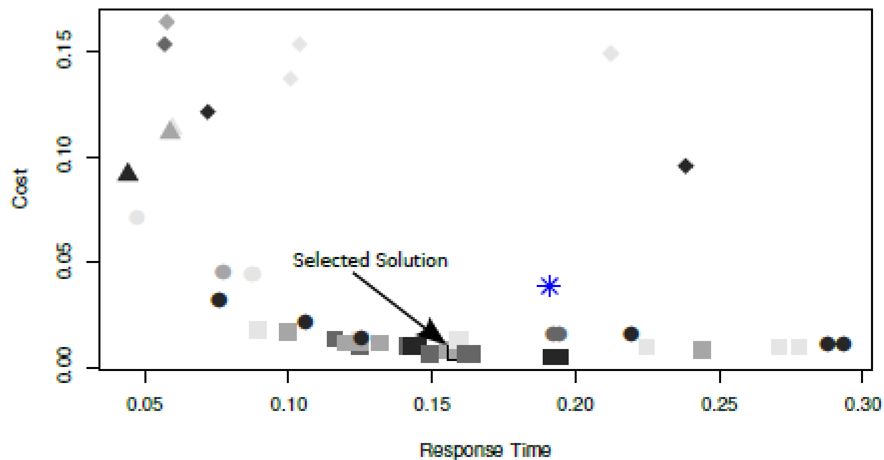

Figure 6: Example of a Pareto front of optimized solutions for Cost and Response time

Chapter 6: Methodology

The methodology of this work consists of code contributions, such as the addition of parsers, the implementation of three algorithms, the addition of various helper methods and the modification of some existing code. Moreover, it consists of modifications of the framework's meta-model, as well as of the creation of the case study and feature model. Certain third party libraries were used for the creation of the feature model and for visualization purposes.

In particular, the extension of the AQOSA tool required: 1) the modification of its workflow process, in order to add a feature modeling step and to repeat the optimization process for every configuration, 2) the use and integration of feature modeling tools, 3) the modification of the framework's intermediate representation (IR) to include features inside the model, 4) commonality analysis, as a last step in the process, in order to find common solutions in all Pareto fronts.

6.1 New process

This work introduces a new workflow process for the tool, which is shown in Figure 7. The new process includes the following steps:

1. Feature Modeling
2. Architecture Modeling
3. Mapping of features to software components
4. Selecting the configurations (products) for optimization
5. Optimization for every product
6. Commonality Analysis

Step 1 is the step where the user has to create the feature model of the product line. The modeling can be done with certain feature modeling tools, which are described in the next section. Step 2 regards the modeling of the system architecture with its software and hardware components and it is not different from the previous process version. The third step is about connecting the features from the feature model to the software components of the architecture. Step 4 is optional and it is used when optimization is needed for specific products only. Step 5 is the ordinary AQOSA optimization tweaked to run for every defined product and producing the respective result sets. Step 6 is the last step of this process and it is where the commonality analysis takes place. The algorithms try to find the solutions that are common in all the result sets and display them.

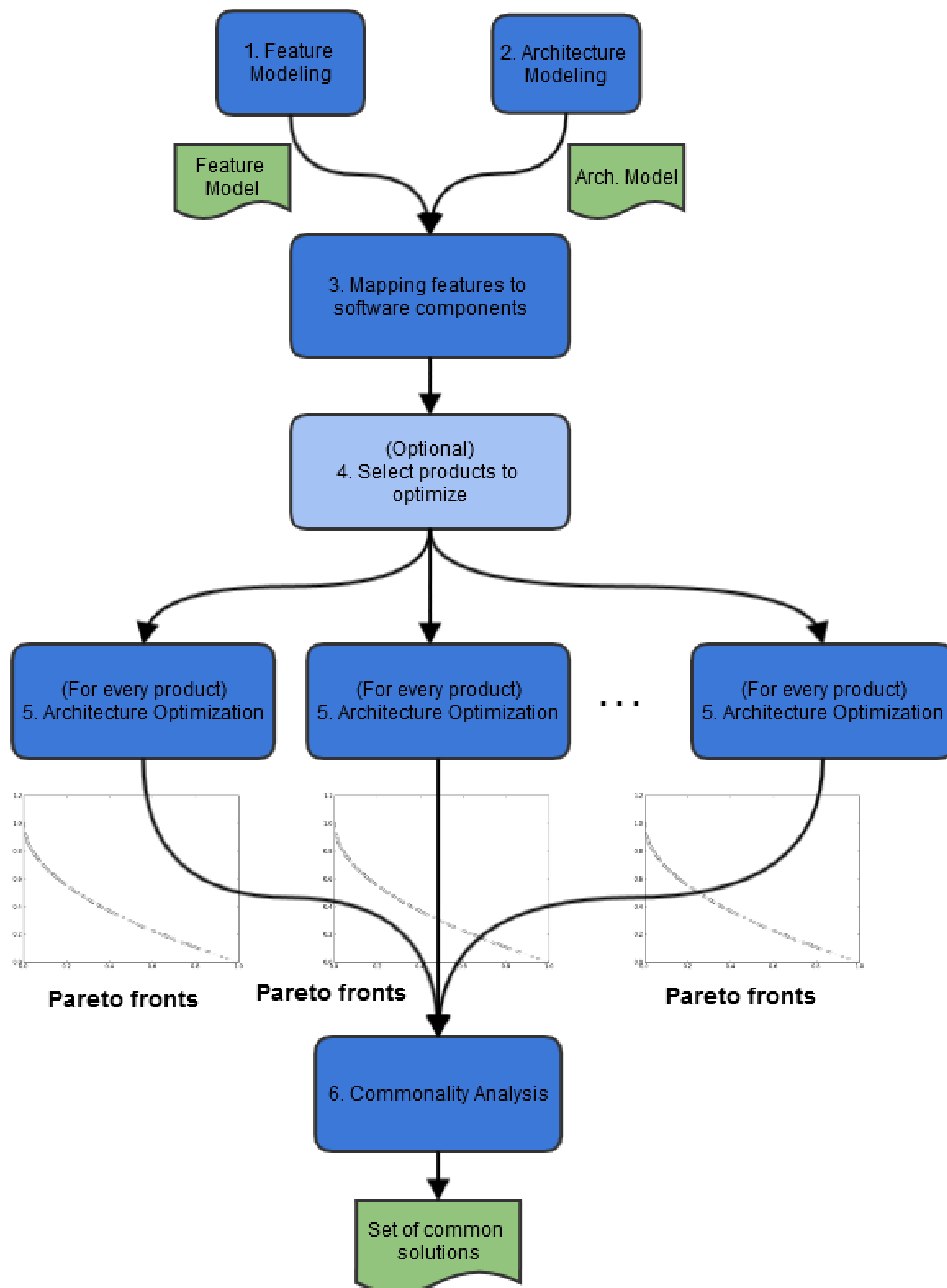

Figure 7: The new proposed process of AQOSA

6.2 Feature modeling tools

Step 1 of the new process requires some tools to model the feature model and find all of its different configurations. After trying several tools, the ones chosen were: Clafer and Alloy. These tools found to be the easiest to use, work great together, have good documentation and can be handled by Java code without having to implement anything (parsers etc.). The only problem is that Clafer cannot be used from inside a Java program, so the user has to write the feature model externally in a file, compile it to Alloy, and then feed this file to AQOSA to find the different configurations with the Alloy Analyzer. However, the way of modeling a feature model is out of the scope of this work and other methods could be used. Figure 8 shows the feature model, written in Clafer, that was used in the case study and whose corresponding diagram is depicted later in Figure 13.

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

Figure 8: Representation, in Clafer, of the feature model used in the case study

6.3 How to link features

The AQOSA framework works by using an internal representation of the system architecture under test. Therefore, a user models their architecture (software and hardware components) based on AQOSA's meta-model, which can be done using the Eclipse's EMF editor. The question now becomes: how this architecture can connect with the features from Alloy?

The solution is to allow the user to model the same features in AQOSA, while modeling the architecture. The first step was to extend the original meta-model with the notion of features that should be modeled in a hierarchical way (parent-child relation). This allows the user to add the same features, as the Clafer model, in the architecture and relate them to appropriate software components. Practically, the only features that can relate to components are the leaf features so this work considers only them for its calculations. The new meta-model is shown in Figure 9.

The second step was to create a map between the Alloy features and the features modeled in AQOSA. For the sake of simplicity, this work enforces that the Alloy and AQOSA features should be named the same but allows the use of different case of letters for clarity reasons. This work uses lower-cased letters for Alloy features and upper-cased letters for AQOSA features. The need of this mapping is essential because the Alloy Analyzer produces the different configurations using the Alloy features, but the optimization work of AQOSA is done using the model, based on its internal representation.

Figure 9: The new AQOSA meta-model showing the addition of features and their relations

6.4 Optimization for multiple products

Next, there was the need to make AQOSA generate alternative architectures for every selected feature configuration. This was done by running the optimization process as before, one time for every configuration. The tool simply keeps the alternative architectures that their software components were strictly related to the features of the current configuration, while filtering out the rest. After the end of the optimizations, the tool produces one set of alternative software architectures (Pareto front) for every selected feature configuration. These fronts are the input of the Commonality Analysis step and its algorithms.

6.5 Commonality analysis of solutions

In order to answer the research question: which software architecture solution can support all the given feature configurations, two algorithms have been created which can find common solutions among the Pareto fronts. The one is called "minimum distance" algorithm and the other "given (fixed) delta" algorithm. The equality or commonality of the solutions is measured by their hardware differences, which is called *distance*. The tool uses a distance algorithm, as the core of the two aforementioned algorithms, to compare two solutions and therefore it will be explained first. The AQOSA tool, by its nature, produces solutions that can vary a lot from each other and it is rare for two solutions to have exactly zero distance. For this reason, two solutions are considered almost equal when they vary within some distance score, Δ (delta).

6.5.1 Solutions distance algorithm (DA)

Each solution object contains a list of CPUs and a list of buses that connects them. Two solutions are considered equal when they do not require any hardware changes in order to have the same hardware. On the other hand, two solutions are not equal when they require some change of hardware in order to match the hardware of the other. This number of changes required from one solution to the other is the distance. Our algorithm is inspired by the Levenshtein distance algorithm (Dictionary of Algorithms and Data Structures, 2013), which measures the distance between two words / strings. The Levenshtein distance algorithm is used in:

- Spell checking
- Speech recognition
- DNA analysis
- Plagiarism detection

Our approach is simpler, because the order of the hardware in the list does not matter, unlike the aforementioned algorithm where the order of letters does matter. The DA works by comparing if each element of one list is contained in the other list and rules them out. The number of the remaining elements is the simple distance, because they show how many of them need to be changed.

In addition, DA considers how these changes are going to be applied. A change can be implemented with one of the following operators: substitution, addition and removal. The costs of these operators are not the same because changing a component with another is always easier than modifying (add/remove) an existing topology, which can also lead to modification of the buses and their connections. Therefore, the algorithm adds weights to the operators so that the cost of Substitution operator is always lower than the cost of an Addition plus a Removal operator. In particular, this work adds to the addition 3 units of difficulty while the Substitution has 1. The algorithm's pseudo code follows below.

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

Figure 10: Pseudo code of distance algorithm

The algorithm works as follows:

1. Find which list is longer than the other
2. Create a copy of each list. (e.g. longList, shortList)
3. For every number in the longList search if it is contained in the shortList
4. If it is contained then delete it from both lists
5. After the loop, the distance is the remaining numbers in the longList

In step 3, the iteration of the long list and the search in the short list is convenient, because the other way around would require to search again in the long list for the remaining numbers.

Example

Solution1 is based on processors with power [10, 20, 30] and solution2 is based on processors with power [10, 10, 20, 40]. In this case the distance is 4 because two operations are needed in order to convert one list to the other (1 substitution and 1 addition). From the point of view of

solution1, 1 CPU change and 1 additional CPU are required and from the point of view of solution2 it is required 1 removal and 1 substitution. Similarly, the distance score can be calculated for the list of buses that connect the CPUs. The final distance score of the solution is the summary of these two scores.

Based on the previous example:

	Iteration 1	**Iteration 2**	**Iteration 3**	**Iteration 4**
Long list	[~~**10**~~, 10, 20, 40]	[~~10~~, **10**, 20, 40]	[~~10~~, 10, ~~**20**~~, 40]	[~~10~~, 10, ~~20~~, **40**]
Short list	[~~10~~, 20, 30]	[~~10~~, 20, 30]	[~~10~~, ~~20~~, 30]	[~~10~~, ~~20~~, 30]

Table 1: Example showing the steps of the distance algorithm working on two lists of processors

6.5.2 Sum of minimum distances algorithm (SMDA)

The concept of this algorithm is to find the total number of minimum steps needed (distance) for one solution to match the some other solution in every other Pareto front. The solution that has the smallest sum of distances can be considered whether it can satisfy the research question or not. In other words, one has to find the minimum distances between one Pareto front and all the others.

Practically, this means that one has to find the minimum distance between one solution and another solution of another Pareto front. Doing the same for all the other Pareto fronts, it sums all the minimum distances and stores the sum in a list. At the end, it sorts the list and one can see if the smallest sum of minimum distances is acceptable. The pseudo code is given in Figure 11.

The algorithm works as follows:

1. For each solution in a Pareto front
2. Calculate and store its distance from each and every solution in another Pareto front
3. Choose and store the minimum distance
4. Do the same (step 2 and 3) for the remaining Pareto fronts
5. After all Pareto fronts are iterated (step 4), sum the minimum distances
6. Store the sum in a final list
7. Repeat step 1 for all Pareto fronts
8. Sort the final list in ascending order.

The outcome of this algorithm is a sorted list of distances, which means that one solution can be common for all Pareto fronts within X number of hardware changes. It is up to the user to decide whether the distance is too much for the solution to be considered common or not.

```
Input: all Pareto fronts (FrontSets)
Output: a sorted list of solutions' total distance
/* for every Pareto solution                                    */
foreach Front1 in FrontSets do
    foreach Solution s1 in Front1 do
        /* check the rest Pareto solutions                      */
        foreach Front2 in the other FrontSets do
            foreach Solution s2 in Front2 do
                calculate distance between s1 and s2;
                store distance in list;
            end
            find the minimum distance in the list;
            store the minimum distance in another list;
        end
        calculate the sum of the minimum distances in the list;
        store the sum in the final list;
    end
end
sort the final list from min to max;
return the final list
```

Figure 11: Pseudo code of SMDA

Example

The algorithm works by iterating over all Pareto fronts (A, B, C for brevity) and checking all solutions (A1, A2, B1, B2, C1).

A1 B1 C1
A2 B2

At first it will iterate over Pareto A and will check every solution. For example A1 with all solutions from the next Pareto front. It will fist measure the distance between A1-B1 (store this result) and then A1-B2 (store result). When it is done with Pareto B then it will find the minimum of these distances. Let's say that the minimum is A1-B1 with distance 0. If there are other Pareto fronts it will check them subsequently and add all these minimum distances between A1 and the other Pareto fronts. In this case, it is only A1-C1 (i.e. distance 1), which is the minimum distance between Pareto A and C, and it is added to the previous minimum distance. So in the final list it stores the sum of minimum distances of the pairs A1-B1 and A1-C1, which is 1. Then it does the same for A2. It checks A2-B1 and then A2-B2. Let's say the minimum here is A2-B2 with distance 1, and then A2-C1 again with distance 1. Their sum is stored in the final list as A2-B2 and A2-C1.

Then it does the same for Pareto B. It checks B1-A1 and then B1-A2, which we have already measured from previous iteration of Pareto A. So again the minimum here is B1-A1 (distance 0). Then it checks the remaining Pareto C with B1-C1 having distance 1 and the sum (0+1) is stored in the final list. For B2 similarly we have B2-A2 as the minimum and the same process for Pareto C.

At the end of all iterations we have the final list:
[(A1-B1 & A1-C1), (A2-B2 & A2-C1), (B1-A1 & B1-C1), (B2-A2 & B2-C1), …]
or showing by total distance:
[1,2,1,2, …]
after sort:
[1,1,2,2, …]

6.5.3 Common solutions within given delta (GDA)

This algorithm is simpler and faster, but requires a Δ value to be manually set before running. The concept is to find a solution which is equal in all Pareto fronts or that can differ within a range (Δ). As discussed earlier, it is rare for two solutions to be exactly equal so one can allow a small deviation in their distance and consider them sufficiently equal, same as double numbers comparison. The pseudo code is displayed below.

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

Figure 12: Pseudo code of GDA

The algorithm works as follows:

1. For every solution in the first Pareto front only
2. Check if it is common with any solution of the next Pareto front (distance <= delta)
3. If common is found then continue searching in the next Pareto front and repeat step 2 and 3 until no other fronts exist. If no common is found then do not bother checking the next fronts and repeat step 1 for next solution.

4. If a solution is common in all fronts then store it in a list

The product of this algorithm is a list of solutions that are common in all Pareto fronts within a deviation delta. This algorithm does not bother to check the rest of the fronts if no common solution is found between a solution and one Pareto front because the commonality is broken. The SMDA checks all the possible combinations regardless of delta, making it slower than GDA. This algorithm can be used when an upper boundary for commonality is needed or when there is sufficient domain and empirical knowledge to determine the appropriate delta deviation.

Example

Suppose that the allowed delta is 1 and that there are 3 Pareto fronts with solutions as follows:

A1 B1 C1
A2 B2

GDA will iterate over solutions in Pareto A and for every solution it will check in the next Pareto (B) for common solutions. At first it will check A1-B1 (say distance here is 2) and then A1-B2 (distance here is 3). No common solution is found so it does not check Pareto C. For A2 it checks A2-B1 (distance 2) and then A2-B2 (distance 1). This is a common solution so now it can check Pareto C. A2-C1 (distance 0) shows another common solution. At the end A2 is stored in the final list of common solutions.

6.5.4 Weaknesses

The DA is currently using only the processors' distance and not using the bus distance at all. This is approach is chosen mostly for demonstration and validation purposes of this work, as it is easier to see which specific changes are required by a solution. The addition of the bus distance only increases the distance score but makes it harder to see which change operators apply. For example a distance score of 3, could be interpreted as 3 CPU substitutions, or 3 bus substitutions, or 1 CPU addition, or 1 bus addition, etc., but using only the CPU distance it would mean only two things. This algorithm could be improved by explicitly providing a list of operators for every distance score.

Another concern for the same algorithm is the fact that it assumes that the calculation of CPU and bus (hardware) distance is sufficient to compare two architectures. Further study is needed to investigate whether some other parameters should also be considered, such as the software components per CPU. The reason this work does not consider software components is that it assumes that a software change, in an architecture solution, is fairly easier and cheaper compared to a hardware change.

The SMDA contains a small bug, though it is not very difficult to fix. As one can notice in the example of that algorithm, the problem is that both ones and both twos refer to the same combination. So it calculates and displays them two times. It is easier to be noticed if one tries to read the example without having the third Pareto C. At the end of all iterations the final list

would be: [(A1-B1), (A2-B2), (B1-A1), (B2-A2)] but A1-B1 and B1-A1 is the same combination / distance. It's like it calculates the permutations of the solutions while it only need the combination, where order does not matter. It's an annoying but not grave error. The solutions would be to implement some kind of history log where all the searched combinations are stored.

6.6 Conclusion

This chapter described the work of this thesis; how the tool was extended, what modifications were made and what tools were used. The work introduced a new process for the tool, which involves the new steps of feature modeling, mapping features with software components, running the optimization for the selected products and performing commonality analysis on the results. In order to model features, two external tools were used and three algorithms were created for finding the common solutions. At the end of the chapter there is a discussion about the identified weaknesses of the algorithms.

Chapter 7: Case Study

7.1 Forming the case study

The feasibility of the proposed method was tested using a case study. Since this work is an extension of a previous work, it was naturally based on the old case study. This way the results are coherent, easily comparable and easy to validate. The old case study is a real-world study from the automotive industry (Etemaadi, et al., 2013) but it does not provide neither many nor interesting feature configurations. For this reason, another case study from the automotive industry (González-Huerta, 2012), which uses many features and cross-tree constraints, was integrated to the old case study. The new features that were added include: cruise control system, park assist system, airbag system, etc. and they make the optimization problem more challenging and complex.

The goal is to run the optimization procedure for all the defined products (feature configurations) in the product line and find the optimal solutions that are common in all products, within the minimum changes.

The feature model of this study is shown in Figure 13. The root element of this model is a car, since this product line targets the automotive industry. Any car product is mandatory to have an ignition switch, interior light and a dashboard. It may optionally have: airbag system, park assist system, theft alarm, cruise control system, stability control system, traction control system and anti-lock breaking system. Some features may come in different variations, for example: the ignition switch can only be either key ignition or button ignition; the dashboard can be either simple or extended. The airbag system, if chosen, must always provide front airbags and optionally side and/or passengers' airbags. The cruise control can either be basic, adaptive or fully adaptive. The stability control system can either be of basic or extended form. There are also cross-tree constraints marked by green and red arrows in the figure. Green arrows show dependency constraints and red arrows show mutually exclusive features (i.e. button ignition with simple dashboard).

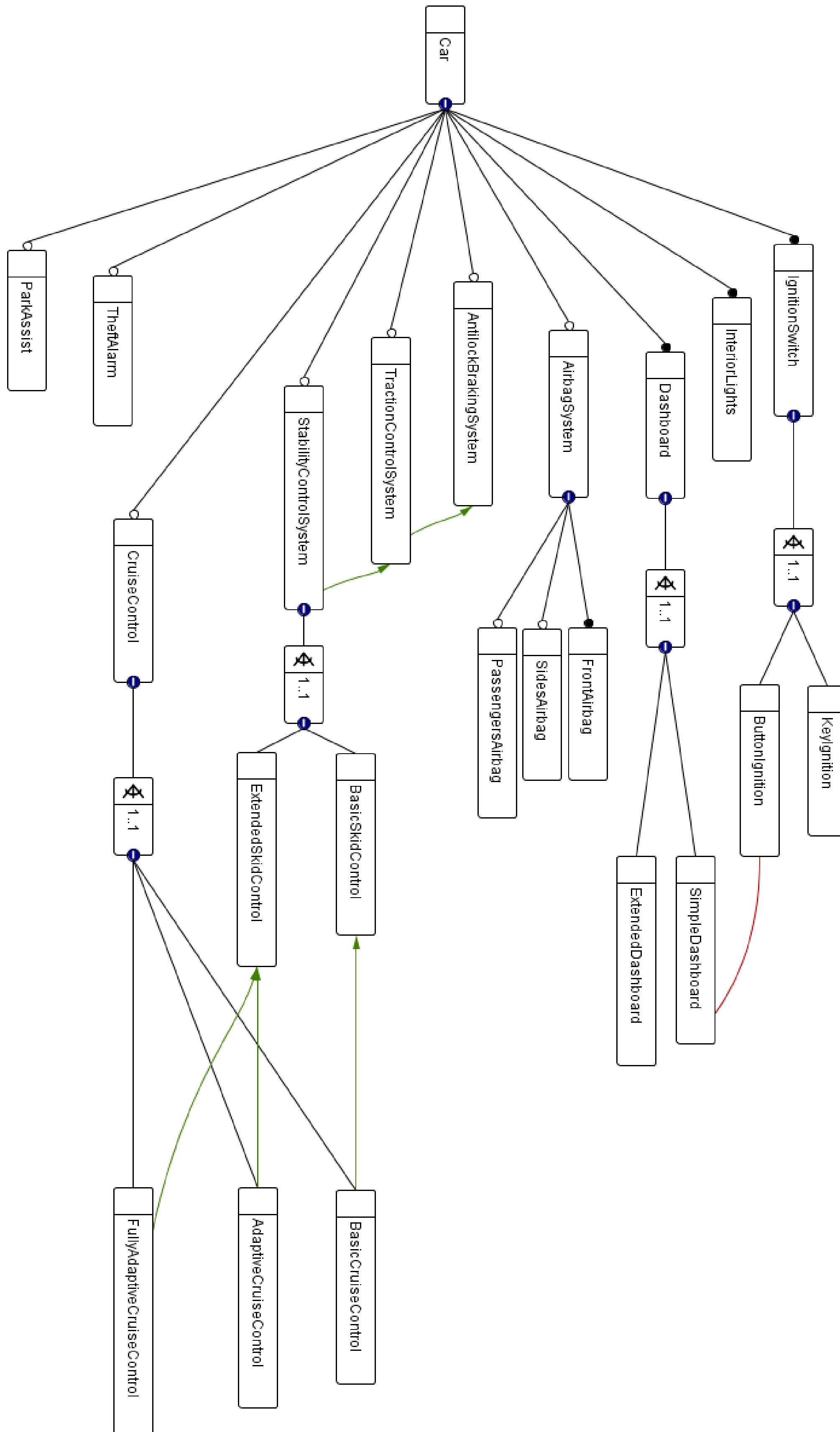

Figure 13: The feature model of the case study

The features are mapped on the software components under the following schema:

Feature	Component
Interior Lights	SC_InteriorLights
Theft Alarm	SC_TheftAlarm
Park Assist	SC_ParkAssistSystem
Key Ignition	SC_IgnitionKey
Button Ignition	SC_IgnitionButton
Simple Dashboard	ProvidePowerModeInfo
	ControlEngineSpeedGauge
	ReadWheelSpeedSensors
	ControlWheelSpeadSensors
	EngineVehicleInterface
	ControlVehicleSpeedGauge
	Gauge_Engine
	ControlOdometer
	Display_Engine
	ReadDriverDoorAjarSwitch
	ReadTripStemButton
Extended Dashboard	ControlCoolantTempGauge
	ControlEngineSpeedGauge
	ControlGearSelectedIndication
	ControlOdometer
	ControlOutsideAirTemp
	ControlVehicleSpeedGauge
	ControlWasherLevelIndication
	ControlWheelSpeed
	Display_Engine
	EngineVehicleInterface
	Gauge_Engine
	ProvidePowerModeInfo
	ReadDriverDoorAjarSwitch

	ReadLowWasherLevel
	ReadOATSensor
	ReadTripStemButton
	ReadWheelSpeedSensors
	TransmissionVehicleInterface
Antilock Braking System	SC_ABS
Traction Control System	SC_TractionControlSystem
Basic Skid Control	SC_BasicSkidControl
Extended Skid Control	SC_ExtendedSkidControl
Basic Cruise Control	SC_BasicCC_CalculateVelocity
	SC_BasicCC_ComputeDesiredSpeed
	SC_BasicCC_ComputeThrottleSetting
	SC_BasicCC_InControl
Adaptive Cruise Control	SC_AdaptiveCC_CalculateVelocity
	SC_AdaptiveCC_ComputeBrakingSetting
	SC_AdaptiveCC_ComputeDesiredSpeed
	SC_AdaptiveCC_ComputeThrottleSetting
	SC_AdaptiveCC_InControl
Fully Adaptive Cruise Control	SC_FullyAdaptiveCC_CalculateVelocity
	SC_FullyAdaptiveCC_ComputeBrakingSetting
	SC_FullyAdaptiveCC_ComputeAirbagSetting
	SC_FullyAdaptiveCC_ComputeDesiredSpeed
	SC_FullyAdaptiveCC_ComputeSeatbeltSetting
	SC_FullyAdaptiveCC_ComputeThrottleSetting
	SC_FullyAdaptiveCC_InControl
Front Airbag	SC_AirbagSystemFront
Sides Airbag	SC_AirbagSystemSides
Passengers Airbag	SC_AirbagSystemPassengers

Table 2: Relation of features with the software components

In this table someone can see the whole set of the software components used in this case study. The old components are the ones which correspond to *simple* and *extended dashboard*,

while the new ones have the prefix *SC*. However, AQOSA works by simulating information flow across components, which cannot be seen in Table 2. Several flows were created - at least one for every feature – and they are best illustrated in sequence diagrams. One flow example can be seen in Figure 14 which depicts the *Fully Adaptive Cruise Control* calculation. One can see how the different software components communicate with each other and with the input and output devices. Extra timing constraints, important to AQOSA, are also visible in the diagram. For brevity, the different input and output devices where grouped.

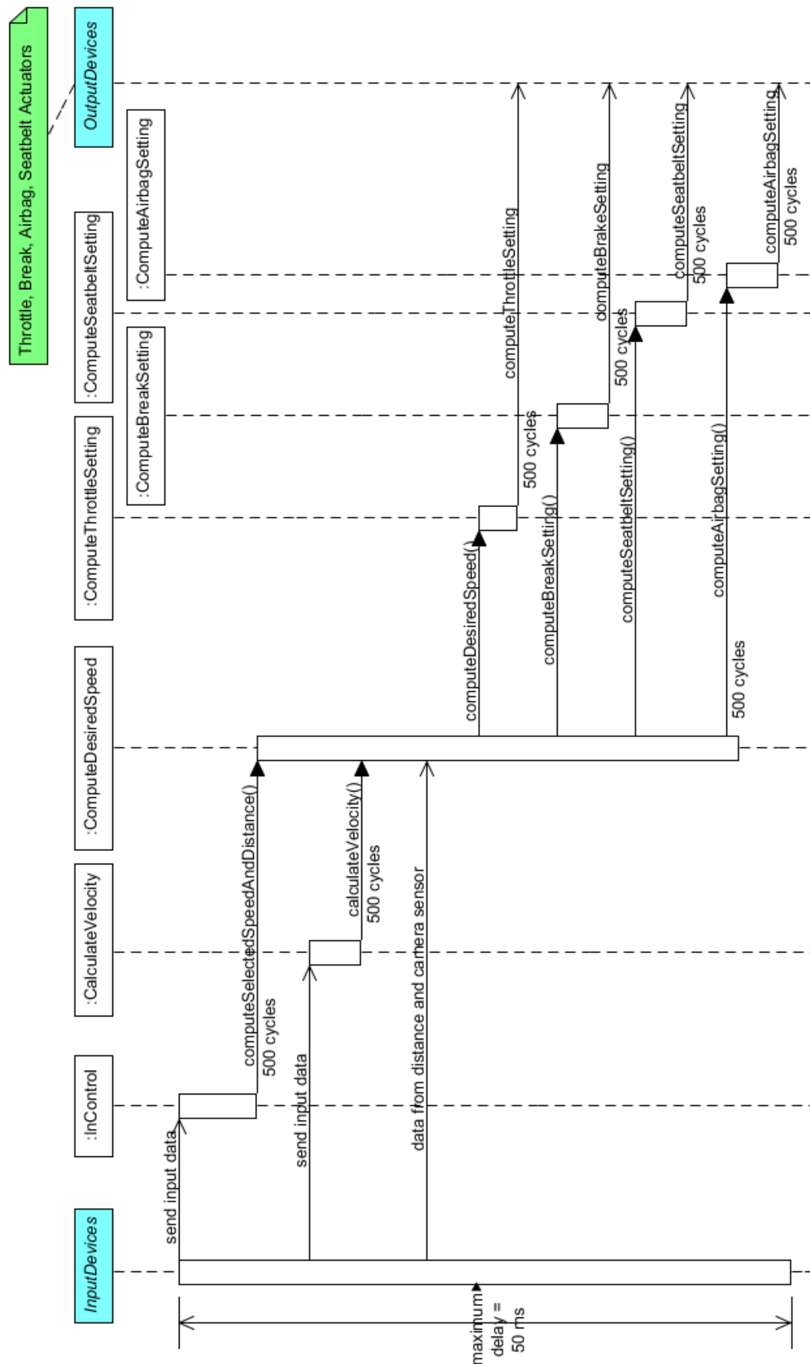

Figure 14: Sequence diagram of FullyAdaptiveCruiseControl system operation

7.2 Products

The feature model allows for 480 different feature configurations that satisfy its constraints. Although this number is considered low for real-world industrial cases, it is prohibiting for running the optimization for all of them since one optimization may take hours. For this reason, only five different cars (products) were selected to be optimized. The five cars were selected in a way that they evenly and gradually cover the whole spectrum of features without being irrational or infeasible. The selection was also based on their resource needs. The tool calculated for all configurations, the total processing power and bandwidth their hardware components need. Then, the configurations were sorted based on those values and five were picked to cover the whole range of values. The claims are displayed in frequency graphs in Figure 15 and Figure 16. The results show that the five selected cars rank 5, 104, 264, 389 and 480 respectively. Feature-wise, the cars vary from simple low-end car to full-featured luxury car, as can be seen next.

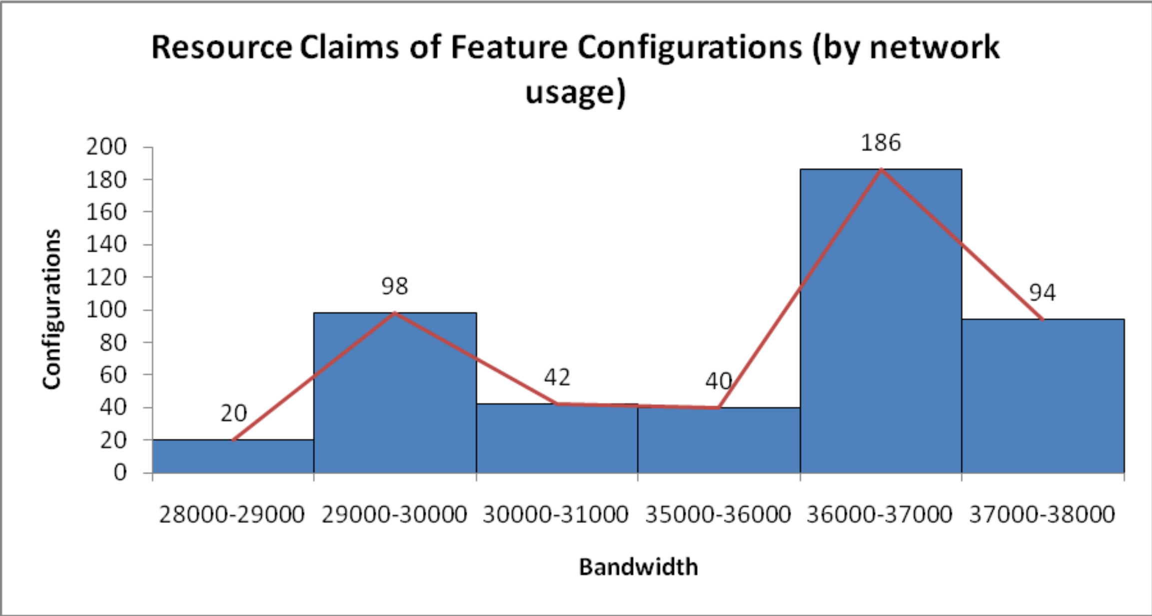

Figure 15: Frequency graph of configurations classified by the bandwidth needs of their related components

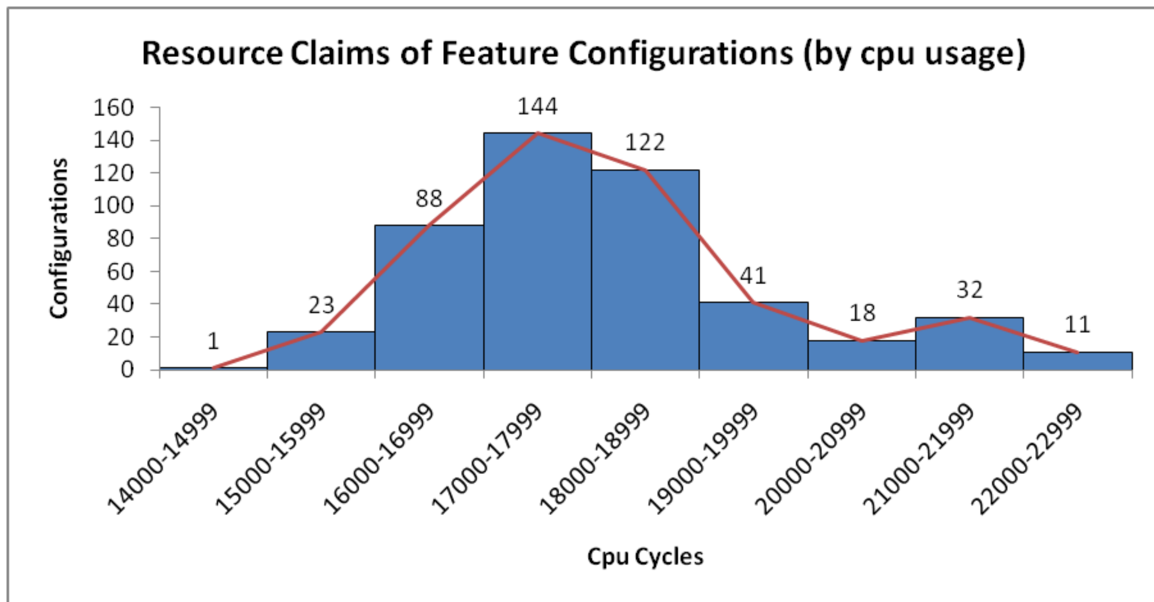

Figure 16: Frequency graph of configurations classified by the processing needs of their related components

7.2.1 Car1

The first selected car does not include many features and its feature configuration is displayed in Figure 17. Like every car from that product line, it contains the mandatory features: interior lights, ignition switch and dashboard. Car1 is a simple car so it comes with a simple dashboard and a key ignition because the simple dashboard excludes the use of button ignition. Moreover, it has the optional feature airbags, with just front airbag deployment.

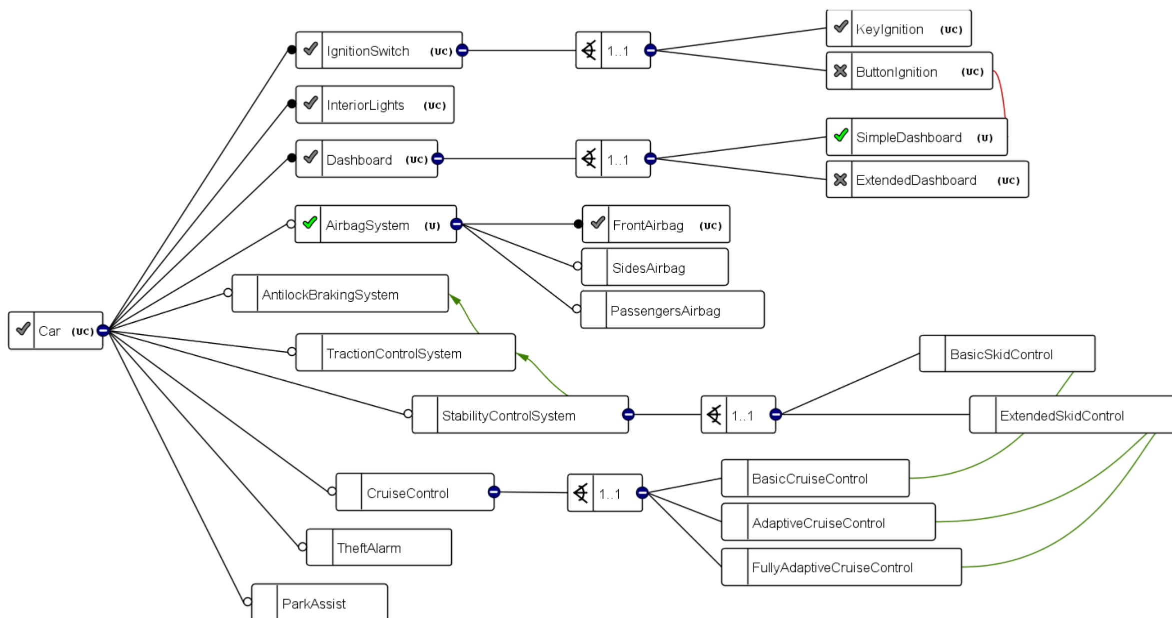

Figure 17: Selected features of Car 1

7.2.2 Car2

Car2 extends Car1 with the addition of the following features: basic skid control, traction control and anti-lock braking system. Figure 18 describes this configuration.

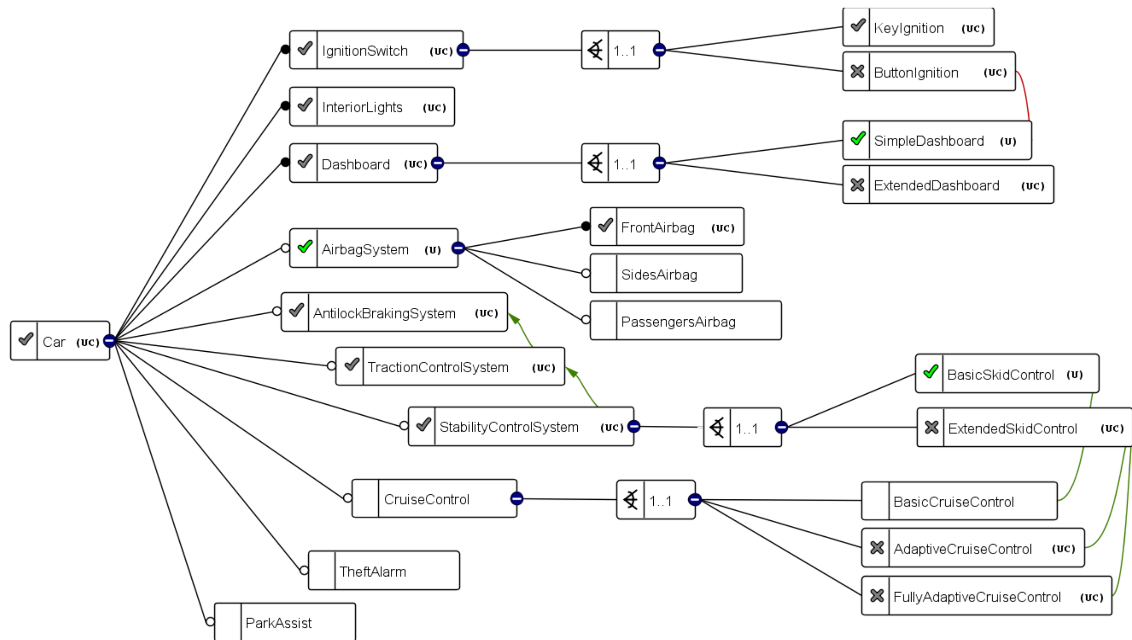

Figure 18: Selected features of Car 2

7.2.3 Car3

Car3 differs from Car2 by having extended dashboard with key ignition and by adding the basic cruise control feature, as can be seen in Figure 19.

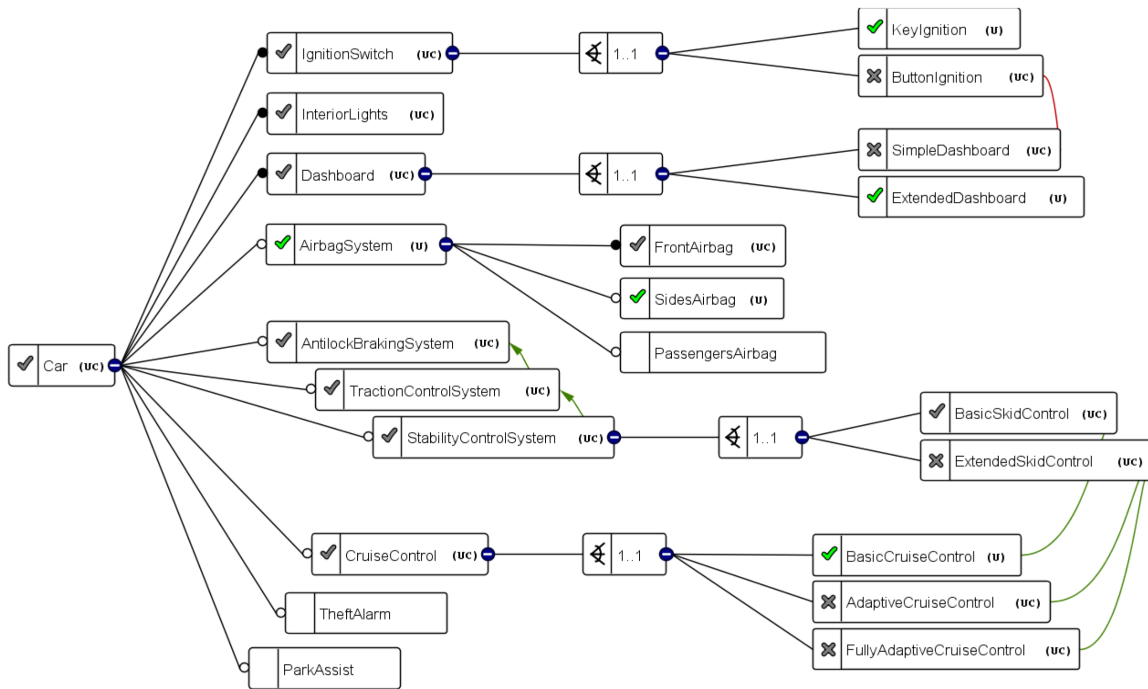

Figure 19: Selected features of Car 3

7.2.4 Car4

This is a more luxurious car because it extends Car3 by adding theft alarm and switching to button ignition and adaptive cruise control. Its feature configuration is displayed in Figure 20.

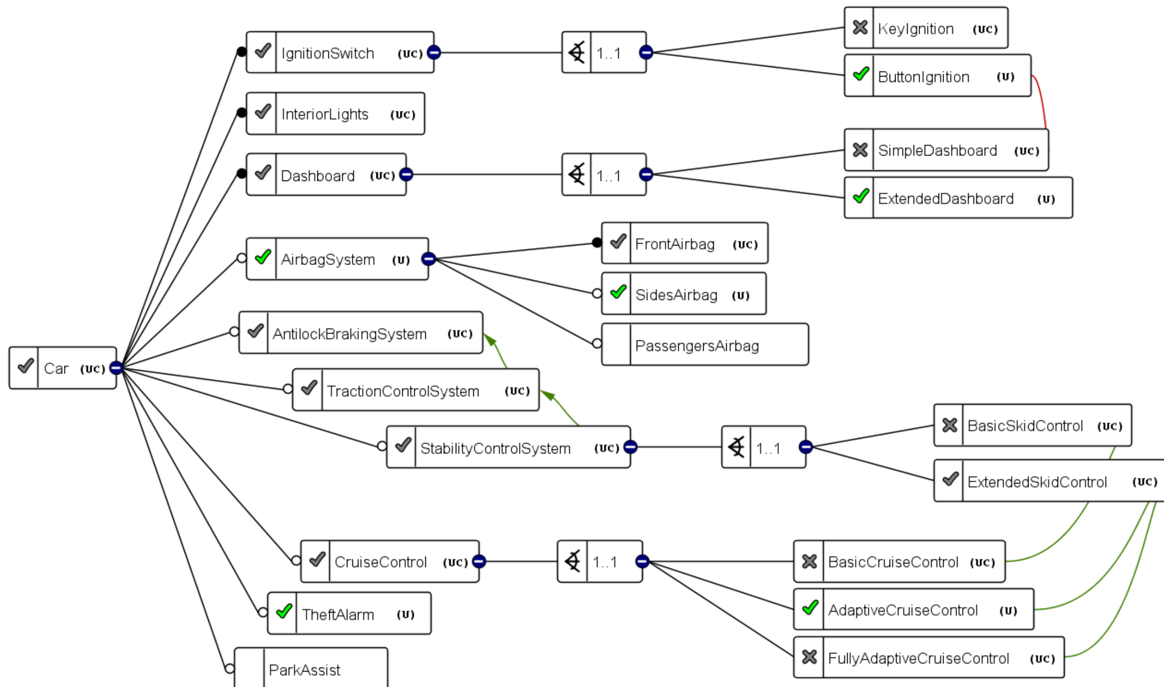

Figure 20: Selected features of Car 4

7.2.5 Car5

Car5 is the most feature-rich car of the line. In Figure 21, it is shown that it implements all the features and selecting the best ones in mutually exclusive cases.

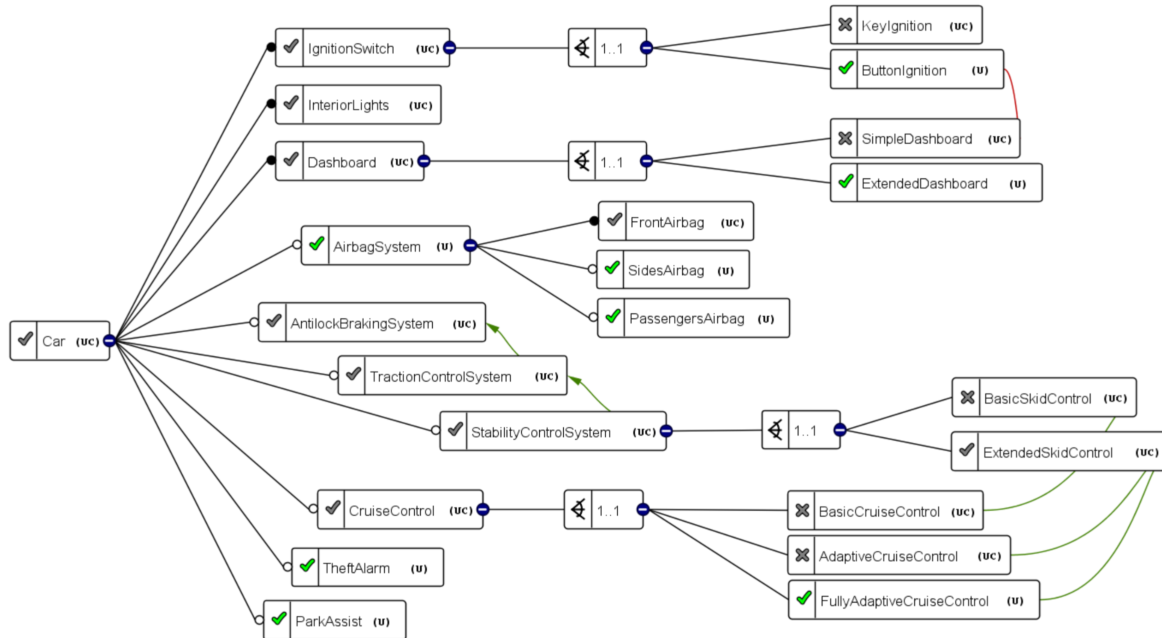

Figure 21: Selected features of Car 5

7.3 Experiment

In order to test the proposed method, an experiment needs to be run for optimization and commonality analysis later. AQOSA works by targeting quality objectives, so in order to run the experiment these objectives have to be set. This experiment considers five quality objectives:

1. Response time
2. Bus utilization
3. CPU utilization
4. Cost
5. Safety

In addition to the software components defined in the case study, the hardware components needed are based on the old case study (Etemaadi, et al., 2013) but tweaked in order to support more software components:

- 10 Processors: 10, 40, 60, 80, 100 MIPS with various failure rates (safety). The cost increases with the CPU power and / or with safety.

- 4 Buses: 10, 33, 125 and 500 kbps. Their latencies: 50, 16, 8 and 2 ms. The cost increases with the bandwidth.

AQOSA ran 30 times based on NSGA-II algorithm with the following parameters:

- Initial population size = 2000
- Parent population size = 100
- Number of offspring = 50
- Archive size (max number of optimized solutions per iteration) = 25
- Number of generations = 500
- All quality objectives were aimed to be minimized.

After the optimization, the commonality analysis was run for all Δ values [0-9]. The GDA was configured with different weights for the change operators. w = 1 for the substitution operator and w = 3 for the addition operator.

7.4 Results

The experiment was run for all the five products and produced 5 different Pareto fronts, one for every product. Each front contained the 25 most optimal solutions for its product, as limited by the experiment parameters, making a total search space of 125 solutions.

After the optimizations, AQOSA performed the commonality analysis on the results. It tried to find common solutions across all five configurations using the GDA. The commonality analysis performed for every delta value between [0 – 9]. The algorithm found the following number of common solutions per delta value:

Delta	0	1	2	3	4	5	6 - 9
Solutions	0	6	14	20	21	21	22

Table 3: Number of common solutions found by GDA in various delta values

This means that GDA did not find any architecture that is exactly the same (distance-wise) in all fronts. However, it found 6 architectures that are close enough to be considered common. The following excerpt (Figure 23), from the results of GDA, shows that solution 3, from Car 1, has distance 0 from solutions of Car 2, Car 4 and Car 5 and distance 1 from solution of Car 3. This shows that the algorithm can find nearly true common solutions which can be left to the architect for further consideration.

Next, the AQOSA ran the SMDA to find the minimum total number of changes of the common solutions. Figure 24 shows an excerpt of the best results of SMDA. The algorithm produced a sorted list of sums of minimum distances. It managed to find solutions that need, in total, less number of changes (distance), than the GDA. In most cases it found at minimum,

a total distance of 1 while the previous algorithm showed only one total distance of 1. A more detailed view is shown in Table 4, which provides a comparison between the two algorithms. The table shows that 3 experiments were run using both algorithms and the total distance of the common solutions found by the two algorithms is displayed. Since the GDA runs many times during an experiment for different delta values, the comparison is done using the number of common solutions found by the lowest possible delta against the equivalent best SMDA solutions. It is obvious that SMDA always finds closer common solutions than GDA.

	Experiment 1		Experiment 2		Experiment 3	
	GDA	SMDA	GDA	SMDA	GDA	SMDA
	4	2	5	1	4	1
	3	2	5	1	4	1
	4	2	5	1	1	1
	3	2	5	1	4	3
	4	2	5	1	4	3
	3	2	4	1	4	3
	4	2				
AVG	3.571429	2	4.833333	1	3.5	2

Table 4: Comparative results of the total distance of the best common solutions

7.4.1 Interpretation of results

AQOSA displays every solution as a series of lists and numbers, as in Figure 22 below. Every solution consists of a list that shows on which processor the various software components are deployed. In this example the first component is deployed to processor 1, the second and third component to processor 2 and so on. Moreover, the tool displays a summary and a description of the processors, a summary and a description of the buses that connect the processors and lastly a boolean mask-table showing which processors the buses connect.

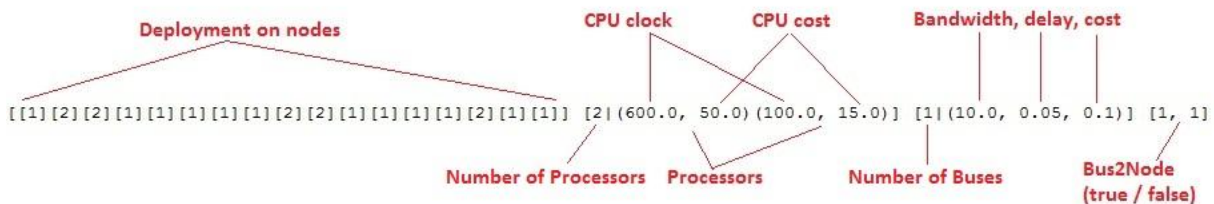

Figure 22: Explanation of an optimized architecture (solution) as displayed by AQOSA

A typical output of the algorithms (Figure 23) consists of a numbered solution, which is the current common solution, followed by its related solutions in the other Pareto fronts in the indented lines. The related solutions have the distance from the common solution as a prefix and adding these distances results in the total distance of the common solution. In this example, one can see that the first common solution differ by distance 1 from the second

solution, in the second Pareto front, distance 1 from the third, etc., and total distance is 1+1+1+1 = 4

```
***Common Solutions: within 1 delta:
Number - Distance with previous - Solution
1)      [[2][1][1][1][1][2][2][1][1][2][2][2][1][1]] [2|(400.0, 50.0)(100.0, 15.0)] [1|(500.0, 0.002, 1.0)] [1, 1]
        Related Solutions (distance is between the common one)
        1   [[2][2][2][2][1][2][2][2][2][2][1][1][1][1][1][2][1]] [2|(400.0, 50.0)(400.0, 50.0)] [1|(33.3, 0.016, 0.25)] [1, 1]
        1   [[2][1][2][1][2][1][2][1][1][1][1][2][2][2][1][2][2][1][1][2][2][2][1][1][1][1][1][1][2]] [2|(600.0, 50.0)(100.0, 15.0)] [1|(33.3, 0.016,
        1   [[1][2][1][1][2][2][1][2][2][2][2][2][2][1][1][1][1][2][1][1][2][1][1][2][2][2][1][2][1][2][2]] [2|(400.0, 50.0)(400.0, 50.0)] [1|(125.0,
        1   [[1][1][1][2][1][1][2][1][1][1][1][1][1][2][1][1][1][2][1][2][1][2][2][1][2][1][1][2][2][1][1][1][2][2][1]] [2|(800.0, 100.0)(100.0, 15.0)
2)  2   [[1][1][1][1][2][2][1][1][1][2][1][2][1][2]] [2|(1000.0, 120.0)(600.0, 50.0)] [1|(10.0, 0.05, 0.1)] [1, 1]
        Related Solutions (distance is between the common one)
        1   [[2][1][1][1][2][1][1][1][2][1][1][2][1][1][2][2][2]] [2|(600.0, 50.0)(600.0, 50.0)] [1|(10.0, 0.05, 0.1)] [1, 1]
        1   [[2][1][2][1][2][1][2][1][1][1][1][2][2][2][1][2][2][1][1][2][2][2][1][1][1][1][1][1][2]] [2|(600.0, 50.0)(100.0, 15.0)] [1|(33.3, 0.016,
        1   [[1][2][2][1][2][2][1][1][1][1][1][1][2][2][1][2][2][1][2][1][1][1][1][1][1][1][1][1][2][2][1]] [2|(600.0, 50.0)(600.0, 50.0)] [1|(33.3, 0
        1   [[2][1][2][1][1][1][2][2][1][1][1][2][1][2][1][1][1][2][1][2][2][2][1][1][2][2][1][1][1][1][1][2][1][1][2]] [2|(1000.0, 120.0)(800.0, 100.
3)  1   [[2][1][2][1][2][2][2][1][2][2][2][2][2][1]] [2|(600.0, 50.0)(600.0, 50.0)] [1|(10.0, 0.05, 0.1)] [1, 1]
        Related Solutions (distance is between the common one)
        0   [[2][1][1][1][2][1][1][1][2][1][1][2][1][1][2][2][2]] [2|(600.0, 50.0)(600.0, 50.0)] [1|(10.0, 0.05, 0.1)] [1, 1]
        1   [[2][1][2][1][2][1][2][1][1][1][1][2][2][2][1][2][2][1][1][2][2][2][1][1][1][1][1][1][2]] [2|(600.0, 50.0)(100.0, 15.0)] [1|(33.3, 0.016,
        0   [[1][2][2][1][2][2][1][1][1][1][1][1][2][2][1][2][2][1][2][1][1][1][1][1][1][1][1][1][2][2][1]] [2|(600.0, 50.0)(600.0, 50.0)] [1|(33.3, 0
        0   [[1][2][1][2][1][2][1][1][2][2][2][1][1][1][2][1][1][2][2][2][1][1][1][1][1][2][2][1][2][2][1][2][2][2][2]] [2|(600.0, 50.0)(600.0, 50.0)]
```

Figure 23: An extract from the results of GDA showing the 3 closest common solutions

```
All distance sums from min to max:
[1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 5, 5,
Number - Distance with previous - Solution

1)      [[2][1][2][1][2][2][2][1][2][2][2][2][2][1]] [2|(600.0, 50.0)(600.0, 50.0)] [1|(10.0, 0.05, 0.1)] [1, 1]
        Related Solutions (distance is between the common one)
        0   [[2][1][1][1][2][1][1][1][2][1][1][2][1][1][2][2][2]] [2|(600.0, 50.0)(600.0, 50.0)] [1|(10.0, 0.05, 0.1)] [1, 1
        1   [[2][1][2][1][2][1][2][1][1][1][1][2][2][2][1][2][2][1][1][2][2][2][1][1][1][1][1][1][2]] [2|(600.0, 50.0)(100.0
        0   [[1][2][2][1][2][2][1][1][1][1][1][1][2][2][1][2][2][1][2][1][1][1][1][1][1][1][1][1][2][2][1]] [2|(600.0, 50.0)
        0   [[1][2][1][2][1][2][1][1][2][2][2][1][1][1][2][1][1][2][2][2][1][1][1][1][1][2][2][1][2][2][1][2][2][2][2]] [2|(
2)  0   [[2][1][1][1][2][1][1][1][2][1][1][2][1][1][2][2][2]] [2|(600.0, 50.0)(600.0, 50.0)] [1|(10.0, 0.05, 0.1)] [1, 1]
        Related Solutions (distance is between the common one)
        0   [[2][1][2][1][2][2][2][1][2][2][2][2][2][1]] [2|(600.0, 50.0)(600.0, 50.0)] [1|(10.0, 0.05, 0.1)] [1, 1]
        1   [[2][1][2][1][2][1][2][1][1][1][1][2][2][2][1][2][2][1][1][2][2][2][1][1][1][1][1][1][2]] [2|(600.0, 50.0)(100.0
        0   [[1][2][2][1][2][2][1][1][1][1][1][1][2][2][1][2][2][1][2][1][1][1][1][1][1][1][1][1][2][2][1]] [2|(600.0, 50.0)
        0   [[1][2][1][2][1][2][1][1][2][2][2][1][1][1][2][1][1][2][2][2][1][1][1][1][1][2][2][1][2][2][1][2][2][2][2]] [2|(
3)  0   [[2][1][1][1][2][1][1][1][2][1][1][2][1][1][2][2][2]] [2|(600.0, 50.0)(600.0, 50.0)] [1|(10.0, 0.05, 0.1)] [1, 1]
        Related Solutions (distance is between the common one)
        0   [[2][1][2][1][2][2][2][1][2][2][2][2][2][1]] [2|(600.0, 50.0)(600.0, 50.0)] [1|(10.0, 0.05, 0.1)] [1, 1]
        1   [[2][1][2][1][2][1][2][1][1][1][1][2][2][2][1][2][2][1][1][2][2][2][1][1][1][1][1][1][2]] [2|(600.0, 50.0)(100.0
        0   [[1][2][2][1][2][2][1][1][1][1][1][1][2][2][1][2][2][1][2][1][1][1][1][1][1][1][1][1][2][2][1]] [2|(600.0, 50.0)
        0   [[1][2][1][2][1][2][1][1][2][2][2][1][1][1][2][1][1][2][2][2][1][1][1][1][1][2][2][1][2][2][1][2][2][2][2]] [2|(
```

Figure 24: An extract from the results of SMDA, showing the total distance of the common solutions in a sorted list, and the 3 best common solutions below

For clarity, some common solutions of bigger distance are visualized, to see how the DA works and how the string output visually corresponds to an actual architecture. Figure 25 shows the string output of the common solutions and Figures 26-27 the visualization of their architectures. To keep it simple this example shows only the CPU distance, so the only important part of the output is the distance score and the CPU description.

```
Fixed Delta Algorithm (Delta = 3)
3)  3   [[2][1][2][2][2][3][2][1][1][3][3][3][3][2]] [3|(100.0, 15.0)(600.0, 50.0)(100.0, 15.0)] [2|(500.0, 0
        Related Solutions (distance is between the common one)
        2   [[3][1][2]...[1][2][1][2][3][2]] [3|(100.0, 15.0)(400.0, 50.0)(1000.0, 120.0)] [2|(500.0, 0.002, 1
        3   [[2][2][2]...[2][2][1][1][2][2][2][1]] [2|(600.0, 50.0)(100.0, 15.0)] [1|(500.0, 0.002, 1.0)] [1,
        2   [[1][1][1]...[3][3][1][1][3][1][2][3]] [3|(600.0, 50.0)(400.0, 50.0)(800.0, 100.0)] [3|(500.0, 0.0
        3   [[1][1][4]...[4][2][4][2][4][4][3][1]] [4|(1000.0, 120.0)(100.0, 15.0)(100.0, 15.0)(600.0, 50.0)]
```

Figure 25: A string output of a common solution and its related ones, found by GDA

The common solution comes from the optimization, so it refers to Car1. Its hardware architecture is visualized in Figure 26. It can be seen that the three processors (100, 100, 600) match the ones in the string output. The related solutions are depicted in Figure 27.

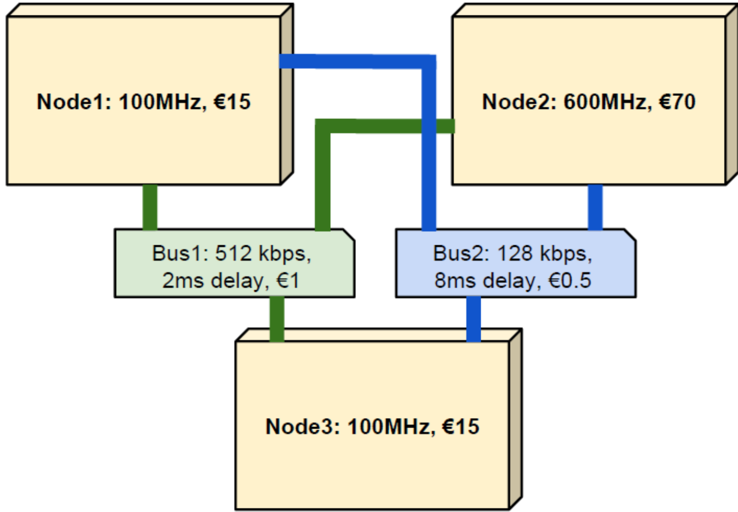

Figure 26: Graphical display of the architecture of the common solution (Car1) in Figure 23. Note the 3 processors of 100, 600 and 100 MHz

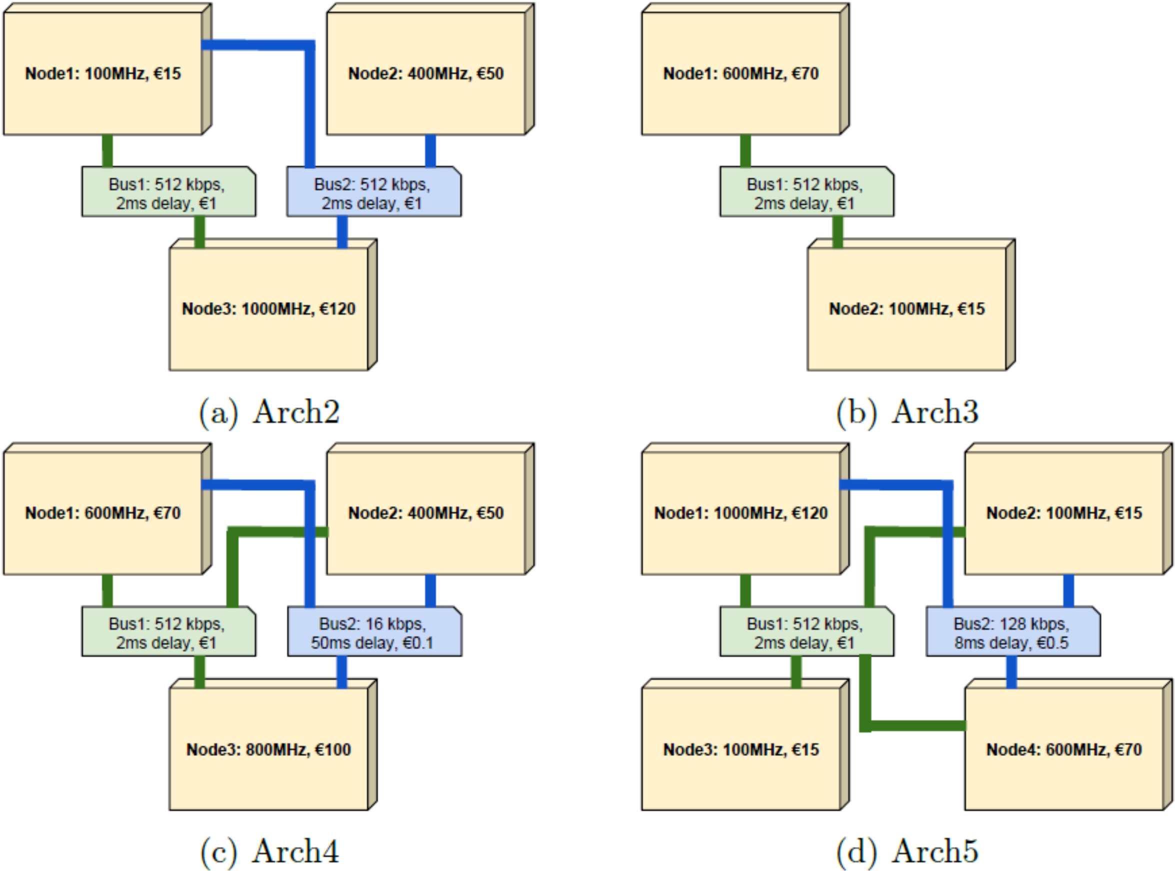

Figure 27: Architectures of a) Car2, b) Car3, c) Car4, d) Car5

Car1 differs from Car2 by distance of 2 because they only need to change 2 CPUs in order to match. Car1 differs from Car3 by distance of 3 because the latter has 2 CPUs in common but lacks the third. So it needs to add another CPU and this operation costs 3 according to the weight. Similarly, the same patterns are can be seen in the other architectures.

The tool also produced JavaScript files, which help visualize the connections between the common solutions and their related ones. It uses the Protovis visualization library (Bostock and Heer, 2009; 2010). Figure 28 shows the common and related solutions of the results of GDA with $\Delta = 1$. Solutions with distance 0 have no lines and solutions with bigger distance have bolder lines.

Figure 28: Common solutions connected to their related solutions. The lines show the distance between them

In conclusion, the results show that the tool finds alternative solutions that are efficient in all their quality attributes, as expected. Moreover, with the extended functionality it finds the solutions that are common in all the selected products of the product line. Even though the common solutions are not exactly equal, they are still sufficiently close to their related ones and answer the research question.

7.5 Limitations

The results of this work show reveal some limitations and threats to its validity. The most obvious problem is that the tool failed to find any common solution with exactly equal (zero distance) from its related ones. Although, it found pretty close distances, it would be nice to see zero distance in some experiment. The problem may lie in the case study simply not having a common solution, or the evolutionary nature of AQOSA, making it improbable to generate 5 identical solutions. Regardless of which problem applies even if there was a zero distance finding which would mathematically prove the case, in a real-world scenario it is unrealistic to expect exactly the same common solution across all configurations. That is due to hundreds of features and thousands of configurations in such cases. Even if there were some common solutions of zero distance, they would be so few that a software architect would still have to choose some that are not exactly zero-distance. In fact, the software

architects expect to choose sufficiently close commons solutions which can be altered with little effort to fit their needs and for this reason this problem can hardly be considered as one.

The most serious threat to validity is that this work has not been reviewed by a domain expert. This case study is an extension of a real-world case study and the parameters of the new components are set intuitively, based on the old components. The original case study was reviewed by experts but this one still needs validation in order to reflect a real-world example. However, it should be noted that this is primarily a proof of concept work which produced evidence that allows it to be applied in more domain-specific problem.

The use of case-studies in this work is also a threat to external validity because it could limit the ability to generalize the findings of this work. Although it is unavoidable for this kind of research, the threat can be minimized by creating a generic design and trying it against many case studies (Torkar, 2012). In that sense, this work provided the functionality and design for the tool to handle any case study. More case studies remain to be seen using this tool.

Another limitation, of practical nature, is that the experiment needs big computational power in order to be executed. The experiment was run on a supercomputer because it was too time-consuming for a laptop. It is noteworthy to mention that on the laptop it was incomplete after 3 hours, but on the supercomputer it was done in slightly more than thirty minutes. The reason is that multi-objective optimization is a computational demanding task by itself. The addition of an extra dimension for features grows the search space even more and makes the tool to execute many times, one time for every configuration under test.

7.4 Conclusion

This chapter presented the case study that was used for testing the proposed approach. The case study is based on the old study used by AQOSA, extended with more features and components. The feature model can give 480 different products, but only 5 were selected because it was too time-consuming (if possible at all) to test them all. The selection was made in a way that the products would be representative of all feature configurations and resource claims. As a final step, the experiment was set up using parameters from the old case study.

Moreover, this chapter presented the results of the experiment. The experiment was run for 5 different products, using the GDA and SMDA to find common solutions. Excerpts from the results of both algorithms were presented, which indicate that both algorithms can find sufficiently close common solutions across all products that apply to all products. However, no identical solutions were found in either algorithm, but it is due to the nature of the optimization process to rarely give identical solutions. Comparing the two algorithms, it is clear that the SMDA gives better results overall than GDA, because it finds related solutions with less total distance. The DA works also fine but whether the hardware distance score is sufficient measure remains to be seen.

Chapter 8: Conclusion

8.1 Summary

The purpose of this work was to generate alternative software architectures of a system, based on its feature model and find those solutions which were applicable to many products. In the past, the software architects could analyze all the components of their architecture and measure its quality characteristics. If it failed to meet the requirements then another architecture had to be optimized by hand. Some tools have been developed to automate the process of finding alternative architectures. Tools, such as AQOSA, use genetic algorithms to optimize an initial architecture towards the quality goals. However, the optimization of these tools regards the software and hardware components but not any features of the system.

This work extended the AQOSA framework with the notion of features, in order to perform feature-based architecture optimization. This extension aims to bridge the gap between the software product line community and the software architecture community, in the sense that the architecture community currently benefits from the existing optimization approaches but not when they apply software product line methods for their systems.

The research question answered by this work is: “Given a feature model, which architecture supports the most features?” The proposed approach was tested on an industrial case study, previously used by AQOSA. The case study was extended with more features, in order to approach the number of configurations used in real cases. The tool itself was modified to be able to model features and run the optimization for all the selected products. Two algorithms were implemented and used for finding common optimal architectures for all products.

The results show that not only the optimizations can find efficient solutions in respect to all quality attributes as before, but also the proposed method identifies similar optimal architectures which are applicable to the range of the selected products in the software product line.

8.2 Future Work

The addition of the extra level of features to the optimization problem makes it even more difficult, since the amount of potential feature combinations can grow very fast. Future work in this area is the use of a smart technique to search the vast search space, because checking every single combination is not efficient. Moreover, a new optimization process could possibly handle more efficiently the problem, as a two-level optimization problem (features and architecture).

The accuracy of the results could be improved if other optimization techniques could be used. The co-evolving Pareto fronts is a technique where optimal solutions could be transferred from one population to another. This could give better or more common solutions among the

Pareto fronts. One other approach would be to use some graph distance algorithm to measure two solutions.

Another possible improvement would be to make the tool run faster, ideally on PCs. As it is now, the tool needs to run on a super computer for medium number of feature configurations. For this reason, it is very difficult to test the tool properly and for software architects to use it. One solution would be to try to parallelize the code, besides using different search techniques.

On another note, more research is needed on some assumptions this work has made. One assumption is that the calculation of hardware distance is sufficient to compare two architectures. This assumption is based on the idea that it is easier to change software allocations on hardware than changing the hardware itself. Further study is needed to investigate whether some other parameters should also be considered, such as software components or even features.

Another assumption is that a conversion of a common solution to its related one, does not affect the distance from the rest of its related solutions. Suppose, for example, that there is the triple of related solutions: (A, B, C) where A is the common solution and its distance from B and C is 1 and 0 accordingly. Nothing guarantees that, when A changes to B, the distance B-C will remain the same.

References

Ai-junkie, (n.d.). *What's the Crossover Rate?* [online]. Available at: <http://www.ai-junkie.com/ga/intro/gat2.html> [Accessed 23 October 2013]

Aleti, A., Björnander, S., Grunske, L. and Meedeniya, I., 2009. ArcheOpterix: An extendable tool for architecture optimization of AADL models. In: *Proceedings of ICSE 2009 Workshop on Model-Based Methodologies for Pervasive and Embedded* Software (MOMPES 2009), May 16 2009, Vancouver, Canada, pp. 61-71.

Alloy: a language and tool for relational models, 2012. *About alloy*. [online]. Available at: <http://alloy.mit.edu/alloy/index.html> [Accessed 23 October 2013]

Antkiewicz, M., Bak, K., Murashkin, A., Olaechea, R., Liang, J.H.J. and Czarnecki, K., 2013. Clafer tools for product line engineering. In: ACM, *Software Product Line Conference Workshops*. Tokyo, Japan, 2013, pp. 130-135. [pdf]. Available at: <http://gsd.uwaterloo.ca/sites/default/files/2013-splc-clafer.pdf> [Accessed 29 November 2013]

Bajpai, P. and Kumar, M., 2010. Genetic Algorithm – an Approach to Solve Global Optimization Problems. *Indian Journal of Computer Science and Engineering*, 1(3), pp. 199-206. [pdf]. Available at: <http://www.ijcse.com/docs/IJCSE10-01-03-29.pdf> [Accessed 03 December 2013]

Bostock, M. and Heer, J., 2009. *Protovis: A Graphical Toolkit for Visualization*. [pdf]. Available at: <http://vis.stanford.edu/files/2009-Protovis-InfoVis.pdf> [Accessed 23 October 2013]

Bostock, M. and Heer, J., 2010. *Protovis: A graphical approach to visualization*. [online]. Available at: <http://mbostock.github.io/protovis/> [Accessed 23 October 2013]

Botterweck, G. and Pleuss, A., 2012. *S2T2-Configurator: interactive support for configuration of large feature models*. [online]. Available at: <http://hdl.handle.net/10344/2586> [Accessed 23 October 2013]

Botterweck, G., Janota, M. and Schneeweiss, D., 2009. A Design of a Configurable Feature Model Configurator. In: *3rd International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS 2009)*. Seville, Spain, 28-30 January 2009. [pdf]. Available at: <http://download.lero.ie/spl/publications/BJS2009Design.pdf> [Accessed 23 October 2013]

Clafer: Lightweight Modeling Language, 2013. *Examples*. [online]. Available at: <http://www.clafer.org/p/about.html> [Accessed 23 October 2013]

Clements, P.C. and Northrop, L., 2001. *Software Product Lines: Practices and Patterns*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

EMF Feature Model, 2013. *Eclipse*. [online]. Available at: <http://www.eclipse.org/proposals/feature-model/> [Accessed 01 April 2013]

Etemaadi, R., Lind, K., Heldal, R. and Chaudron, M.R.V., 2013. Quality-Driven Optimization of System Architecture: Industrial Case Study on an Automotive Sub-System. *Journal of Systems and Software (JSS)*, 86 (10), pp. 2559-2573.

Etxeberria, L. and Sagardui, G., 2008. Variability Driven Quality Evaluation in Software Product Lines. In: *Proceedings of 12th International Software Product Line Conference*, pp. 243-252.

Goldberg, D., 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley: Boston, MA, USA.

González-Huerta, J., 2012. *Integration of Quality Attributes in Software Product Line Development*. Tesis de Máster en Ingeniería del Software, Métodos Formales y Sistemas de Información (MISMFSI) Grupo de Ingeniería del Software y Sistemas de Información (ISSI) Departamento de Sistemas Informáticos y Computación (DSIC) Universidad Politécnica de Valencia (UPV).

Grunske, L., Lindsay, P., Bondarev, E., Papadopoulos, Y. and Parker, D., 2007. An Outline of an Architecture-Based Method for Optimizing Dependability Attributes of Software-Intensive Systems. In: *Architecting Dependable Systems IV*, pp. 188-209. Springer: Berlin.

INFORMS Computing Society (ICS), 2010. The Nature of Mathematical Programming. In: *Mathematical Programming Glossary*. [online]. Available at: <http://glossary.computing.society.informs.org/index.php?page=nature.html> [Accessed 23 October 2013]

Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E. and Peterson, A.S., 1990. *Feature-oriented domain analysis (FODA) feasibility study*. [pdf]. Available at: <http://www.sei.cmu.edu/reports/90tr021.pdf> [Accessed 29 November 2013]

Karlsruhe Institute of Technology, 2012, Paladio Feature Model (1.0 stable). [computer program] Karlsruhe Institute of Technology. Available at: <http://sdqweb.ipd.kit.edu/wiki/Palladio_Feature_Model#Installation_.26_Resources> [Accessed 01 April 2013]

Koziolek, A., Koziolek, H. and Reussner, R., 2011. PerOpteryx: Automated Application of Tactics in Multi-Objective Software Architecture Optimization. In: ACM SIGSOFT conference -- QoSA and ACM SIGSOFT symposium -- ISARCS on Quality of software architectures -- QoSA and architecting critical systems – ISARCS (QoSA-ISARCS 2011), Boulder, Colorado, USA, 20 June 2011 - 24 June 2011, ACM, pp. 33-42

Lee, K., Kang, K.C. and Lee, J., 2002. *Concepts and Guidelines of Feature Modeling for Product Line Software Engineering*. [pdf]. Available at:

<http://pdf.aminer.org/000/366/067/concepts_and_guidelines_of_feature_modeling_for_product_line_software.pdf> [Accessed 23 October 2013]

Li, R., Etemaadi, R., Emmerich, M.T.M. and Chaudron, M.R.V., 2011. An Evolutionary Multiobjective Optimization Approach to Component Based Software Architecture Design. In: *Evolutionary Computation (CEC), 2011 IEEE Congress on 5-8 June 2011*, pp. 432-439.

Lukasiewycz, M., Glaí, M., Reimann, F. and Teich, J., 2011. Opt4J: a modular framework for meta-heuristic optimization. In: ACM, Krasnogor, N. and Lanzi, P.L. (Eds.), *Genetic and Evolutionary Computation Conference (GECCO)*, pp. 1723-1730.

Mendonca, M., Branco, M., Cowan, D., 2009. S.P.L.O.T. - Software Product Lines Online Tools. In: *Companion to the 24th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA 2009)*, October 2009, Orlando, Florida, USA.

Pieterse, V. and Black, P.E. eds., 1999. Algorithms and Theory of Computation Handbook, Levenshtein distance. In: *Dictionary of Algorithms and Data Structures*. CRC Press LLC 22 August 2013. [online]. Available at: <http://www.nist.gov/dads/HTML/Levenshtein.html> [Accessed 23 October 2013]

Prebys, E.K., 2007. The Genetic Algorithm in Computer Science. *MIT Undergraduate Journal of Mathematics*, pp. 165--170.

Rylander, B. and Foster, J., 2001. Computational Complexity and Genetic Algorithms. In: *Proceedings of the World Science and Engineering Society's Conference on Soft Computing, Advances in Fuzzy Systems and Evolutionary Computation*, pp. 248-253. [pdf]. Available at: <http://people.ibest.uidaho.edu/~foster/Papers/45648.pdf> [Accessed 23 October 2013]

Segura, S., 2009. *A feature diagram representing a configurable e-shop system*. [image online] Available at: <http://en.wikipedia.org/wiki/File:E-shopFM.jpg> [Accessed 01 December 2013]

Software Engineering Institute, 2013. *Software Product Lines – Overview*. [online]. Available at: <http://www.sei.cmu.edu/productlines/> [Accessed 01 December 2013]

Torkar, R., 2012. Empirical Software Engineering Course: *Validity Threats*. Chalmers University of Technology, University of Gothenburg. Original slides contributed by Mikael Svahnberg (BTH).

Whitley, D., 1994. A genetic algorithm tutorial. *Statistics and Computing*, 4(2), pp. 65–85. Kluwer Academic Publishers: s.l.

[*One Point Crossover*] 2013 [image online]. Available at: <http://en.wikipedia.org/wiki/File:OnePointCrossover.svg> [Accessed 02 December 2013]

Wikipedia, 2013. *Mutation (genetic algorithm)*. (3 March 2013) [online]. Available at: <http://en.wikipedia.org/wiki/Mutation_%28genetic_algorithm%29> [Accessed 23 October 2013]

Yin, R. K., 1994. *Case Study Research: Design and Methods* (2nd ed.). Newbury Park: Sage Publications.