Measuring the Evolution of Automotive Software Models and Meta-Models to Support Faster Adoption of New Architectural Features

Darko Đurišić

University of Gothenburg

Division of Software Engineering
Department of Computer Science & Engineering
University of Gothenburg
Gothenburg, Sweden, 2015
Measuring the Evolution of Automotive Software Models and Meta-Models to Support Faster Adoption of New Architectural Features

Darko Đurišić

Copyright ©2015 Darko Đurišić except where otherwise stated. All rights reserved.

Technical Report No 127L
ISSN 1652-876X

Department of Computer Science & Engineering
Division of Software Engineering
University of Gothenburg
Gothenburg, Sweden

This thesis has been prepared using \LaTeX. Printed by Chalmers Reproservice, Gothenburg, Sweden 2015.
To my loving Wife,
whose love and support is my constant source of inspiration,
and
to my Mom & Dad,
without who I would not be the person that I am today.
Abstract

**Background:** The ever-increasing amount of software in cars today combined with high market competition demands fast adoption of new software solutions in car development projects. One challenge in enabling such a fast adoption is to develop the architecture and models of the automotive software systems in a structured and controlled way.

**Objective:** The main objective of this thesis was to enable the fast utilization of new architectural features in automotive software models. This was achieved by developing methods and tools to analyze the evolution of the domain-specific meta-models that are used to define the language of software models and their features. In particular, we wanted to identify the underlying changes caused by meta-model evolution related to a specific set of architectural features and assess their impact on both the architectural models and modeling tools used by different roles (e.g., the Original Equipment Manufacturers, OEMs, and their suppliers) in the development process.

**Method:** We achieved our objective by conducting an action research project in close collaboration with the Volvo Car Group (Volvo Cars) and the consortium of the AUTOSAR standard, which aims to standardize the architecture of automotive software systems. This collaboration facilitates fast feedback from experts in the field on the problems, ideas and methods we developed in the course of this research, thereby enabling the validation of the research results and proposed methods in on-going development projects, i.e., their direct application in the industry.

**Results:** We identified the most suitable software measures for measuring the evolution of both the automotive software models and meta-models. The calculation and presentation of the measurement results were done with the support of two, newly-developed tools. We also developed a method for the automated identification of an optimal set of new architectural features that should be adopted in development projects to facilitate the decision-making process concerning the selection of which of these new features would be adopted.

**Conclusion:** We applied the developed methods and tools to the architectural models and meta-models used at Volvo Cars and concluded that they provide valuable input for the decision-making process concerning which new versions of the standardized meta-model should be used in different projects. We also concluded that these methods and tools can facilitate the assessment of the impact of adopting new architectural features on the different roles involved in the development process.
First of all I would like to express my deepest gratitude to my supervisor Miroslaw Staron. I was very fortunate to get a chance to work with Miroslaw since my Master study days. This served as an initial spark that lit my desire to pursue a PhD. Miroslaw’s guidance and constant feedback before and during my PhD project was invaluable for my development as a researcher.

Next I would like to thank my co-supervisors Matthias Tichy and Jörgen Hansson whose ideas and comments significantly improved the quality of my research and included publications. I would also like to thank my managers at Volvo Stefan Andreasson and Hans Alminger for steering my development as an engineer and for supporting my wish to become an industrial PhD student.

I am very grateful to Urban Kristiansson, now retired Senior Technical Advisor from Volvo, for guiding me through the process of becoming a part of Volvo’s Industrial PhD Program and ensuring my project funding. I am also very grateful to my colleagues Nicklas Fritzson, from the AUTOSAR team, and Ola Reiner, from the VSEM team, who kept this project alive with their ideas about the research and the application of the research results at Volvo.

Finally my inexpressible appreciation goes to my family who is the most important thing in my life. First and foremost I want to thank my wife Bojana for unlimited encouragement in my moments of doubt and for sharing my moments of success. I hope that I will find a way to make it up to you for all the work you had to carry out yourself during my traveling and writing. I am also very grateful to my parents Slobodan and Mirjana whose support in my personal and professional life decisions was an important source of motivation.

The research presented in this thesis was conducted within the QuaSAR@car research project which is funded by Volvo Cars, as part of the Volvo Industrial PhD Program (VIPP), and by Swedish Governmental Agency for Innovation Systems (VINNOVA), under the grant no. 2013-02630.
List of Publications

Included papers

This licentiate thesis is based on the following papers:


Other publications

The following paper is published but not appended to this thesis due to the content overlap with the first paper.

# Contents

Abstract v

Acknowledgment vii

List of Publications ix

1 Introduction 1

1.1 Modeling and meta-modeling 2

1.1.1 Domain-specific meta-models 5

1.1.2 Domain-specific modeling tools 6

1.1.3 Architectural models and features 7

1.2 Automotive software development 7

1.2.1 AUTOSAR modeling environment 9

1.2.2 AUTOSAR meta-model 10

1.2.3 AUTOSAR models 12

1.3 Software measurement 12

1.3.1 Measurement process 14

1.3.2 The use of software measurement in this thesis 16

1.4 Research questions and contributions 17

1.4.1 Paper contributions 19

1.4.2 Industrial contributions 20

1.5 Research methodology 22

1.5.1 Theory of action research 22

1.5.2 Theory of case study 23

1.5.3 One action research cycle as a Case study 25

1.5.4 Research validity 27

1.6 Future work 28

1.7 Personal contribution 28

2 Paper A 33

2.1 Introduction 34

2.2 Related Work 36

2.3 Research Method 37

2.4 Designing Software Systems at VCC 40

2.4.1 Logical View 40

2.4.2 Deployment View 42

2.5 Quality Metrics 43

2.5.1 Logical View Measures 44
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5.2</td>
<td>Deployment View Measures</td>
<td>48</td>
</tr>
<tr>
<td>2.6</td>
<td>Presentation and Interpretation of Results</td>
<td>49</td>
</tr>
<tr>
<td>2.6.1</td>
<td>Presentation of Measurement Results</td>
<td>49</td>
</tr>
<tr>
<td>2.6.2</td>
<td>Interpretation of Measurement Results</td>
<td>50</td>
</tr>
<tr>
<td>2.7</td>
<td>Example</td>
<td>52</td>
</tr>
<tr>
<td>2.7.1</td>
<td>The Example System Description</td>
<td>52</td>
</tr>
<tr>
<td>2.7.2</td>
<td>Measurements and Results Presentation</td>
<td>55</td>
</tr>
<tr>
<td>2.7.2.1</td>
<td>Logical View</td>
<td>55</td>
</tr>
<tr>
<td>2.7.2.2</td>
<td>Deployment View</td>
<td>58</td>
</tr>
<tr>
<td>2.7.3</td>
<td>Results Interpretation</td>
<td>60</td>
</tr>
<tr>
<td>2.8</td>
<td>Validation of the Metrics</td>
<td>61</td>
</tr>
<tr>
<td>2.8.1</td>
<td>Theoretical Validation</td>
<td>61</td>
</tr>
<tr>
<td>2.8.2</td>
<td>Empirical Validation</td>
<td>63</td>
</tr>
<tr>
<td>2.9</td>
<td>Conclusions</td>
<td>68</td>
</tr>
<tr>
<td>3</td>
<td>Paper B</td>
<td>75</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction</td>
<td>76</td>
</tr>
<tr>
<td>3.2</td>
<td>Case Study Context</td>
<td>77</td>
</tr>
<tr>
<td>3.3</td>
<td>Related Work</td>
<td>79</td>
</tr>
<tr>
<td>3.4</td>
<td>Case Study Design</td>
<td>80</td>
</tr>
<tr>
<td>3.5</td>
<td>Case Study Results</td>
<td>83</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Identified roles</td>
<td>84</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Mapping of roles</td>
<td>84</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Relevant types of changes</td>
<td>85</td>
</tr>
<tr>
<td>3.5.4</td>
<td>Considered releases</td>
<td>86</td>
</tr>
<tr>
<td>3.5.5</td>
<td>Measurement results</td>
<td>87</td>
</tr>
<tr>
<td>3.6</td>
<td>Discussion and Validation of the results</td>
<td>89</td>
</tr>
<tr>
<td>3.7</td>
<td>Conclusions</td>
<td>90</td>
</tr>
<tr>
<td>4</td>
<td>Paper C</td>
<td>95</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>96</td>
</tr>
<tr>
<td>4.2</td>
<td>Related Work</td>
<td>97</td>
</tr>
<tr>
<td>4.3</td>
<td>AUTOSAR Meta-Model and its Role</td>
<td>98</td>
</tr>
<tr>
<td>4.4</td>
<td>Case Study Design</td>
<td>99</td>
</tr>
<tr>
<td>4.5</td>
<td>Definition of the Metrics</td>
<td>104</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Number of classes</td>
<td>104</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Number of attributes</td>
<td>104</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Depth of inheritance</td>
<td>105</td>
</tr>
<tr>
<td>4.5.4</td>
<td>FanIn</td>
<td>105</td>
</tr>
<tr>
<td>4.5.5</td>
<td>FanOut</td>
<td>106</td>
</tr>
<tr>
<td>4.5.6</td>
<td>FanInOut</td>
<td>106</td>
</tr>
<tr>
<td>4.5.7</td>
<td>Package coupling</td>
<td>107</td>
</tr>
<tr>
<td>4.5.8</td>
<td>Coupling between classes</td>
<td>107</td>
</tr>
<tr>
<td>4.5.9</td>
<td>Package cohesion</td>
<td>108</td>
</tr>
<tr>
<td>4.5.10</td>
<td>Cohesion ratio</td>
<td>108</td>
</tr>
<tr>
<td>4.6</td>
<td>Case Study Results</td>
<td>109</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Entire AUTOSAR meta-model</td>
<td>109</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Software Component Template</td>
<td>110</td>
</tr>
<tr>
<td>4.6.3</td>
<td>System template</td>
<td>111</td>
</tr>
</tbody>
</table>
### CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.6.4</td>
<td>Summary and validation of the metrics results</td>
<td>112</td>
</tr>
<tr>
<td>4.6.5</td>
<td>Recommendations</td>
<td>114</td>
</tr>
<tr>
<td>4.7</td>
<td>Conclusion</td>
<td>115</td>
</tr>
</tbody>
</table>

### 5 Paper D

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>122</td>
</tr>
<tr>
<td>5.2</td>
<td>Related work</td>
<td>123</td>
</tr>
<tr>
<td>5.3</td>
<td>Research methodology</td>
<td>124</td>
</tr>
<tr>
<td>5.4</td>
<td>MeFiA method definition</td>
<td>126</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Meta-data model for the changes</td>
<td>126</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Linking meta-model changes to features</td>
<td>127</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Optimizing the set of adopted features</td>
<td>128</td>
</tr>
<tr>
<td>5.4.4</td>
<td>Assumptions for the MeFiA method</td>
<td>130</td>
</tr>
<tr>
<td>5.5</td>
<td>Automotive software development</td>
<td>131</td>
</tr>
<tr>
<td>5.6</td>
<td>Applying MeFiA on AUTOSAR features</td>
<td>133</td>
</tr>
<tr>
<td>5.6.1</td>
<td>Optimization for the entire meta-model</td>
<td>134</td>
</tr>
<tr>
<td>5.6.2</td>
<td>Role-based optimization</td>
<td>136</td>
</tr>
<tr>
<td>5.6.3</td>
<td>Aggregated role-based optimization</td>
<td>138</td>
</tr>
<tr>
<td>5.7</td>
<td>Conclusion and future work</td>
<td>139</td>
</tr>
</tbody>
</table>

### 6 Paper E

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>146</td>
</tr>
<tr>
<td>6.2</td>
<td>AUTOSAR based software development</td>
<td>147</td>
</tr>
<tr>
<td>6.3</td>
<td>Related work</td>
<td>148</td>
</tr>
<tr>
<td>6.4</td>
<td>ARCA tool</td>
<td>149</td>
</tr>
<tr>
<td>6.4.1</td>
<td>The architecture of ARCA tool</td>
<td>149</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Quantifying/presenting the meta-model changes</td>
<td>151</td>
</tr>
<tr>
<td>6.4.3</td>
<td>Presenting the results of software metrics</td>
<td>152</td>
</tr>
<tr>
<td>6.4.4</td>
<td>Presenting/quantifying the feature related changes</td>
<td>153</td>
</tr>
<tr>
<td>6.4.5</td>
<td>Combining all tool’s functionalities in car projects</td>
<td>156</td>
</tr>
<tr>
<td>6.5</td>
<td>Conclusion</td>
<td>156</td>
</tr>
</tbody>
</table>
Chapter 1  
Introduction

The use of software in cars dates from the end of the 1970s and the start of the 1980s when microcomputer applications began to be used for engine control [1]. Today, software plays an important role in almost every function of a modern car. Automotive functionalities, such as adaptive cruise control, the lane departure warning system and automated parking support, depend primarily on the use of software. The analysis of Charette [2] suggests that it takes more than 100 million lines of code to put a premium car on the road today, which is more code than some passenger airplanes require to operate their avionics and on-board support systems. The trend of increasing amounts of software in modern cars is expected to continue [3] due to new, future functionalities, including autonomous drive and car-to-car communication.

In order to stay competitive, car manufacturers (Original Equipment Manufacturers, OEMs) must constantly incorporate new automotive functionalities in their car models. This requires fast software innovation cycles in the car development projects, which in turn require changes in the software used in the actual cars (the automotive software). The automotive software is at a high abstraction level described by the software architectural model; the latter usually must be updated to support the addition of new car functionalities and enable changes in the automotive software. These updates represent the evolution of the automotive software architecture and its model.

The automotive architectural model is comprised of a set of predefined architectural features that represent its building blocks. These features are described by domain-specific meta-models, the availability of which determines the technology that can be used in the cars themselves, for example, the use of Ethernet as a communication medium between distributed parts of the automotive software system. As the automotive software technology evolves, so do the domain-specific meta-models in order to support the new architectural features, which are then used to model different architectural components of the system. This means that the models and the domain-specific meta-models evolve in parallel with each other.

This thesis addresses the problem of cost-efficient management of the domain specific meta-model evolution in car development projects. Our main objective is to facilitate the decision-making process concerning which of the new architectural features will be used in future cars in order to support the
modeling of new functionalities. We achieved this objective by developing tools and methods to automate the analysis of meta-model changes caused by specific architectural features.

The proposed methods are based on a number of software measures that we defined to evaluate the evolution of the architectural models and meta-models. These measures can quantify the changes between different meta-model versions for a set of new architectural features. The measurement results can also be used to visualize the evolution of both the architectural models and meta-models with respect to a number of properties, such as size, complexity and coupling. We used these results to assess the impact of different architectural features on the modeling tools used by various roles (e.g., the OEMs and their suppliers) involved in the automotive software development process.

We applied the proposed methods and tools to the evolution of the Automotive Open System Architecture (AUTOSAR) meta-model [4] and the architectural models developed at Volvo Cars. The AUTOSAR meta-model is a standardized meta-model used to define the language for the architectural models of the automotive domain and is fundamental to the development of automotive modeling tools. We showed that the methods can be used to facilitate the decisions regarding which new releases of the AUTOSAR standard or subsets of its new features will be adopted in on-going or future car development projects.

This chapter is structured as follows: Section 1.1 describes the role of models and meta-models in software development; Section 1.2 describes the automotive modeling environment and the role of AUTOSAR in this environment; Section 1.3 describes both the importance and use of software measurement in this thesis; Section 1.4 defines our research questions and describes the contribution and impact on industry of each paper included in this thesis; Section 1.5 describes the methodology we employed to analyze the data and obtain the results; finally, Section 1.6 summarizes our future research plans.

Chapters 2-6 present the individual papers included in this thesis. Each paper is independent and represents one study that addressed one or several, smaller research questions.

1.1 Modeling and meta-modeling

Modeling plays an important role in the development of large software systems because it can reduce their complexity by raising the abstraction level. This abstraction level is achieved by specifying what the system does rather than how it does this [5]. Models and meta-models are the two most important concepts involved in any modeling environment. Based on the definitions of Bézivian et al. [6], a model represents a simplified representation of a software system that has been created for a specific purpose, whilst a meta-model represents the model specification using a specific modeling language.

The word ”meta” indicates that something was done twice, in this case the modeling; for example, the meta-model represents ”a model of the model” and the meta-meta-model represents ”a model of the meta-model”. Applying this logic several times creates a meta-modeling hierarchy, which is also referred to as the meta-pyramid [7] and is generally accepted to consist of four layers:
1. The **M3 layer**: a meta-meta-model that defines the modeling concepts

2. The **M2 layer**: a meta-model that defines the language specifications

3. The **M1 layer**: a model that defines the instance specifications

4. The **M0 layer**: an object that defines the system instances

These layers are connected by the instantiation mechanism, i.e., each layer represents an instance of the layer above; for example, \( M0 \) is an instance of \( M1 \), except for the top layer \( M3 \), which is considered to be an instance of itself. Generally, one element is an instance of another element if the instantiating element defines the characteristics of that instance element, and the instance element defines the specific details of these characteristics. For example, an instantiating element may specify that a \( \text{Signal} \) has a data-type and an instance of this \( \text{Signal} \) may define an integer data-type.

A model is an instance of the meta-model, whilst a meta-model is an instance of the meta-meta-model. According to the strict meta-modeling principle [8], all model elements on one layer of a meta-modeling hierarchy are instances of the model elements of the layer above except for the top layer. No other relationships that cross the boundaries of different layers are possible. The \( \text{instanceOf} \) relationship, however, is not transitive; for example, the \( M1 \) model element will only receive characteristics from the \( M2 \) meta-model elements and not from the \( M3 \) meta-meta-model elements [9].

The meta-modeling hierarchy described above is also accepted by the Object Management Group (OMG) [10] which is considered a de facto standard for meta-modeling (see Figure 1.1).

**Figure 1.1: The Meta-Object Facility (MOF) layers**

The Meta-Object Facility (MOF) [11] resides at the top of the hierarchy. This is a meta-meta-model that defines the general modeling concepts used by meta-models on the \( M2 \) layer. A frequently used meta-model on the \( M2 \) layer is UML (Unified Modeling Language). The actual UML models reside on the \( M1 \) layer and their actual execution at run-time resides on the \( M0 \) layer.
CHAPTER 1. INTRODUCTION

The $M3$ and the $M2$ layers represent language specifications. Four important concepts of one language specification can be distinguished, namely the abstract syntax, concrete syntax, static semantics (their well-formedness) and dynamic semantics (also referred to as semantics). The abstract syntax describes the structural essence of the language, for example, a general concept used for all purposes. The concrete syntax renders the abstract concepts for a specific purpose (e.g., using graphical notation) and there can be more than one concrete syntax for an abstract syntax. The static semantics impose a set of constraints on the abstract syntax in order to apply the abstract concepts. Finally, the dynamic semantics give meaning to the syntax notation of the language. A meta-modeling environment must at least specify the abstract syntax, but can also specify the concrete syntax and the static and dynamic semantics [12].

Two different types of meta-models, namely the linguistic and ontological meta-models, can be distinguished based on the two forms of the $instanceOf$ relationship described earlier [5]. The primary purpose of the linguistic meta-models is to define a language (the abstract syntax) for the specification of models instantiating this meta-model. The primary purpose of the ontological meta-models is to define the meaning of the models. The ontological meta-models can be used as domain-specific language definitions instantiating a linguistic meta-model to provide both the syntax and static semantics for one modeling environment (please refer to the description of the domain-specific meta-models below).

These two forms of the $instanceOf$ relationship are also responsible for the meta-modeling property known as dual classification [13], where the instances on the $M0$ layer are both ontological instances of the $M1$ classes (which in turn are instances of the $M2$ Class) and linguistic instances of the $M2$ Object. Therefore, the illustration of the meta-modeling layer hierarchy shown in Figure 1.1 breaks the strict schema principle as the $M0$ instances directly instantiate (linguistically) the $M2$ Object, i.e., the $M1$ layer is omitted.

One way of solving this problem is to consider the strict meta-modelling principle for only one of the two forms of $instanceOf$ relationships, i.e. either ontological (from the system modelers perspective) or linguistic (from the tool implementers and language designers perspectives). In terms of the linguistic approach, ontological modeling can, to some extent, be achieved using stereotypes [14, 15]. Another approach would be to represent the linguistic and the ontological instantiations using two dimensions, where the ontological instantiation is depicted horizontally and the linguistic instantiation is depicted vertically; an example is shown in Figure 1.2.

The $O0$ and $O1$ vertical layers represent the ontological layers, while the $L0$, $L1$ and $L2$ horizontal layers represent the linguistic layers. We can see that the $User$ Object on the linguistic layer $L1$ and the ontological layer $O0$ is both a linguistic instance of the generic $Object$ on the $L2$ and an ontological instance of the concrete $User$ Class on the $O1$ layer. Moreover, the actual $Runtime$ object on the linguistic layer $L0$ only occurs on the ontological layer $O0$; thus, we can only form a mental picture of it on the $O1$ layer.

Finally, not all modelers interpret the relationship between the models and the meta-models in this way, as demonstrated by Kühne [16], because it depends on the exact interpretation of the word ”meta”. So far, we have referred
to a meta-model as ”a model of the model using the instanceOf relationship”; for example, the executing code is an instance of the UML class diagram. We can, however, also define a model of the model using the representedBy relationship, where both models reside on the same meta-modeling layer; for example, a UML class diagram is a model of the source code. Figure 1.3 shows an example of the instanceOf and the representedBy relationships.

Figure 1.3: The representedBy and instanceOf relationships

Generally it is agreed that the relationship between the models and the meta-models is based on the instanceOf relationship, whilst the representedBy relationship indicates the model-to-model transformation.

1.1.1 Domain-specific meta-models

Meta-modelling plays an important role in the development of the description languages suitable for modeling of specific domains [17], including automotive, telecommunications and avionics. A domain-specific model rep-
represents an abstract representation of the system of a particular domain, whilst a domain-specific meta-model defines the syntax and the semantics of the domain-specific models instantiating this meta-model [18]. Several domain-specific meta-models may focus on specifying different subject areas [19], such as data, work-flows and tasks.

The syntax of one domain-specific meta-model includes both the abstract and concrete syntaxes, which define the language constructs, and the entities and their relationships within the abstract syntax, respectively. The semantics, however, include only static semantics (e.g., each signal needs to have a data-type), which can be achieved by specifying different constraints, such as the range of the relationships and OCL (Object Constraint Language) constraints [20], to ensure their well-formedness. On the other hand, dynamic semantics (e.g., all signals must meet their timing requirements) cannot be specified in the domain-specific meta-model; usually they are achieved by providing supporting specifications in a natural language. In the case of UML-based, domain-specific meta-models, UML profiles with the defined stereotypes and tag definitions can also be a powerful mechanism of customizing the concrete syntax and static semantics of the UML meta-model for a specific domain.

Any one, domain-specific modeling environment may contain an arbitrary number of layers; in practice, there can be more than the four layers described in Figure 1.1. These layers must evolve to support the addition of new system functionalities, and the evolution of one layer may require the evolution of some or all of the other layers [21]. For example, in order to express new modeling solutions on the $M1$ layer, the $M2$ layer must evolve in order to describe how to model these new solutions. Similarly, evolution of the $M2$ layer may require evolution of existing models of the $M1$ layer in order to maintain conformity between the $M1$ and $M2$ layers.

The challenge of parallel evolution of both the models and meta-models of one domain-specific modeling environment, therefore, must be addressed by system modelers and meta-modelers, although this combined model-meta-model evolution can be automated to a certain extent [22], for example, the automated re-factoring of existing $M1$ models in order to conform to a new $M2$ meta-model [23].

1.1.2 Domain-specific modeling tools

System modelers of one domain-specific modeling environment usually rely on software modeling tools to create and update the models and generate code based on these models. Since the development of large software systems often involves many roles potentially using different modeling tools in the development process, a smooth exchange of models between these roles can be somewhat challenging. Nevertheless, a smooth exchange can be enabled by defining and gaining consensus from all roles for a domain-specific meta-model, which is then fundamental to the development of all tools used in a specific modeling environment. This is based on the assumption that if two modeling tools adopt the same model structure defined by the meta-model, they can exchange a number of software models that comply with this meta-model [24].

Since the modeling tools are based on a commonly-accepted meta-model, its evolution may significantly impact all the tools in a specific modeling en-
1.2 Automotive software development

The development of automotive electrical systems (software and hardware) relies on the use of different models [26] and is usually based on the V-model process shown in Figure 1.4.

![VCC Process Description II](image)

Figure 1.4: The development of automotive electrical systems
The left side of the V-model determines the design of the electrical system, whilst the right side is responsible for system verification. The first step in the development process is to define a model for a number of vehicle functionalities, such as an early collision warning. The system architects then define a general, architectural model of the system by creating a topology of the Electronic Control Units (ECUs) connected via electronic buses and allocating the vehicle functionalities onto these ECUs. The architectural models are refined in the system design step, where a number of software (and hardware) components controlling the execution of vehicle functionalities are modeled and allocated onto different ECUs. These steps and their verification matches on the right side of the “V” are usually done by the OEMs. In this thesis, we focus on the architectural and system models, i.e., on their specific design steps.

The design, implementation and verification of both the software and hardware for one ECU are usually done by the suppliers who are organized into different tiers according to their responsibilities, such as implementation of the software components and the middleware software and hardware delivery. The development process based on the aforementioned V-model is usually iterative and consists of several iterations where new vehicle functionalities are added and the architecture and system designs are updated accordingly.

The AUTOSAR standard was introduced in 2003 as a joint partnership of the OEMs and their suppliers in order to facilitate the development of automotive electrical systems based on the V-model. Today, AUTOSAR consists of more than 150 global partners [4] and, therefore, is considered to be a de facto standard for automotive software system development. Thus, it is hardly feasible these days for OEMs to develop automotive electrical systems that are not based on the AUTOSAR standard because fewer suppliers are available, which significantly increases development costs.

The AUTOSAR standard defines the following main objectives:

1. Increased re-use of architectural components developed by the automotive software suppliers in different car projects (within one or multiple OEMs). This allows cheaper software components to be used with higher quality (as these components are tested in several car projects).

2. Standardization of the middleware and hardware layers, which enables software designers and implementers to focus more on the design and implementation of complex vehicle functionalities.

3. Standardization of the exchange format for the architectural models, which enables a smooth model exchange between a number of software modeling tools developed by different tool suppliers.

These AUTOSAR objectives are achieved using three-layer ECU architecture. The top layer (Application software) defines the software components and their exchange points and is closely associated with the architecture system design steps of the V-model. The middle layer (Run-time environment) controls communication between different software components abstracting the fact that they may be allocated to different ECUs. If they are allocated to different ECUs, transmission of the respective signals on the electronic buses is done by the bottom layer (Basic software), which is generally responsible
for "basic" ECU functionalities, such as communication with the hardware, buses, memory and diagnostic services.

Both the Run-time environment and Basic Software layers are completely standardized by AUTOSAR, such that AUTOSAR provides detailed specifications for all architectural components of these layers. This standardization, together with the clear distinction between the Application software, Run-time environment and Basic software layers, allow the system designers and implementers to specifically focus on the development of vehicle functionalities without being distracted by middleware and hardware design. The application software and basic software architectural components are developed often by different suppliers who specialize in either one of these areas, referred to as Tier 1 and Tier 2 suppliers, respectively.

In order to standardize the exchange format for the architectural models of the Application software layer between the OEMs and the Tier 1 and 2 suppliers, AUTOSAR defines a domain-specific meta-model that specifies the language for these models. Based on the AUTOSAR meta-model, the modeling tools of OEMs and suppliers can exchange the architectural models efficiently, thereby minimizing any tool interoperability issues.

### 1.2.1 AUTOSAR modeling environment

The AUTOSAR modeling environment consists of five layers, the names of which are taken from the AUTOSAR Generic Structure specification [27]:

1. The ARM4: MOF 2.0, e.g., the MOF Class
2. The ARM3: UML and AUTOSAR UML profile, e.g., the UML Class
3. The ARM2: Meta-model, e.g., the SoftwareComponent
4. The ARM1: Models, e.g., the WindShieldWiper
5. The ARM0: Objects, e.g., the WindShieldWiper in the ECU memory

The AUTOSAR modeling environment defines five meta-layers in an attempt to incorporate both the linguistic and ontological `instanceOf` relationships that are of importance to the tool suppliers and the system designers, respectively. This means that subsequently the strict meta-modeling principle is not achieved. To solve this problem, we characterized the AUTOSAR meta-modeling hierarchy using the two-dimensional representation shown in Figure 1.5 and represent the linguistic instantiation (corresponding to the MOF layers) vertically and the ontological layers horizontally.

The ARM2 and ARM1 layers represent the ontological layers O1 and O0, respectively, and both reside on the same linguistic layer L1 (MOF layer M1). The ARM4 and ARM3 layers represent the linguistic layers L3 and L2 (MOF layers M3 and M2), respectively. The ARM0 layer corresponds to the linguistic layer L0 (MOF layer M0) and resides on the ontological layer O0 because it represents the execution of the concrete object in an ECU. This representation conforms to the strict meta-modeling principle as the `instanceOf` relationships never cross the boundaries of more than two layers.

The ARM4 and ARM3 layers define the abstract syntax, while the ARM2 layer (referred to as the AUTOSAR meta-model) defines the concrete UML
syntax and the static semantics of the AUTOSAR models residing on the $M_1$ layer. The AUTOSAR meta-model also uses the AUTOSAR UML profile from the $ARM_3$ layer, which specifies the used stereotypes and tags. All other constraints and dynamic semantics are defined in the natural language specifications that support the AUTOSAR meta-model.

### 1.2.2 AUTOSAR meta-model

The AUTOSAR meta-model ($ARM_2$ layer) is divided into a number of top-level packages, referred to as ”templates”. Each template is standardized and defines how to model one specific part of the automotive electrical system. For example, the `SoftwareComponentTemplate` defines how to model software components and their communication, whilst the `SystemTemplate` defines how to model ECUs (i.e., the allocation of software components onto the ECUs, ECU connections using electronic buses, etc.) and their communication (i.e., the signals exchanged on the electronic buses).

Based on the AUTOSAR templates residing on the $ARM_2$ layer, the OEMs and their suppliers can create proprietary models on the $ARM_1$ layer instantiating these templates for different parts of the automotive system. This is true for all models except for those instantiating the `ECUCDefinitionTemplate`. The latter models contain a definition of the ECU Basic software configuration parameters and are standardized by AUTOSAR. In order words, AUTOSAR provides the standardized $ARM_1$ models for the configuration parameters of all Basic software modules. The actual values of these parameters are defined in the models instantiating the `ECUCDescriptionTemplate` and they reference their corresponding definitions (see the example shown in Figure 1.6).

On the smallest granularity, standardized models of the `ECUCDefinitionTemplate` are divided into a number of packages, where each package contains
1.2. AUTOMOTIVE SOFTWARE DEVELOPMENT

the configuration parameters of one Basic software module. On the highest granularity, these models are divided into different logical packages, including ECU communication, diagnostics, memory access and IO access. Moreover, in papers B, D and E of this thesis, we have analyzed the evolution of the standardized models of the ECUCDefinitionTemplate, together with the evolution of the meta-model templates, and focused on the logical packages of ECU communication (the role of ECU communication configurators) and diagnostics (the role of Diagnostic configurators).

The values of certain configuration parameters from the ECUCDescriptionTemplate models can be automatically derived from models of other templates, such as the SoftwareComponentTemplate or SystemTemplate. This process is called "upstream mapping" and is referred to as this in the thesis papers. The role of Upstream mapping tool developers defined in Paper B, for example, is to implement tool support for the automatic derivation of parameter values of the ECUCDescriptionTemplate models from the SoftwareComponentTemplate and SystemTemplate models.

A simplified example of different AUTOSAR templates (ARM2) and their models (ARM1) is shown in Figure 1.6. The gray color represents the elements standardized by AUTOSAR, whilst the light blue color represents proprietary elements modeled by different OEMs.

![Figure 1.6: An example of the AUTOSAR templates and their models](image)

On the ARM2 layer, we can see the elements of the five templates. The ECUCDefinitionTemplate specifies the modeling of the definition of ECU parameters and containers of these parameters with an example of the integer parameter. The ECUCDescriptionTemplate specifies the modeling of the con-
tainer and parameter values. The `SoftwareComponentTemplate` and the `SystemTemplate` specify a simplified modeling of the signal transmission by the software components. Finally, all elements of the templates shown are inherited from a common element in the `GenericStructureTemplate` (Identifiable), which provides both a name and unique identifier (UUID) for these elements.

The standardized model of the `ECUConfigurationDescription` can be seen on the `ARM1` layer, which shows the `ECUSignal` container with the `InitValue` integer parameter. These two elements both have a tagged value with the name `UM`, denoting Upstream Mapping. The `UM` tagged value for the `ECUSignal` container refers to the `SystemSignal` from the `SystemTemplate`. The `UM` tagged value for the `InitValue` parameter refers to the `initValue` attribute of the `SystemSignal`. This implies that for every `SystemSignal` instance in the `SystemArchitecture` model, one container instance in the `ECUConfigurationValues` will be created with an integer parameter instance. The value of this parameter instance will be equal to the `initValue` attribute of that `SystemSignal` instance (0 in our example). This derivation of the ECU configuration parameter values from the system architectural models can be automated using modeling tool support.

### 1.2.3 AUTOSAR models

Managing the evolution of the AUTOSAR meta-model is essential in enabling the incorporation of new features in automotive architectural models and thus in car development projects. This is because these models must be fully compliant with the AUTOSAR meta-model to ensure a smooth model exchange between a number of roles in the development process. Thus, they are also referred to as the AUTOSAR models. In order to update the AUTOSAR models with new architectural features, all roles (e.g. the OEMs and their suppliers) must update their modeling tools according to new versions of the AUTOSAR meta-model.

The AUTOSAR models are structural models and they should not be confused with behavioral models that are used to describe concrete ECU functionalities, for example, auto-braking when a pedestrian is detected on the vehicle’s trajectory. These behavioral models are usually developed in Matlab Simulink tool from which the actual source code (C code) can be generated. Certain parts of the ECU source code related to the functionality of the middleware layer, `Run-time environment` and the configuration of the `Basic Software` modules (referred to as the aforementioned “upstream mapping” process) can also be generated from the AUTOSAR models.

The AUTOSAR models are expressed using Extensible Markup Language (XML) and are validated by the AUTOSAR XML schema generated from the AUTOSAR meta-model. This process is referred to as the XML Meta-data Interchange (XMI) and is a standardized process for data exchange between different roles in the development process created by OMG [28].

### 1.3 Software measurement

Measurement in software engineering plays an essential role not only to assessing a variety of software system quality attributes, such as reliability,
1.3. SOFTWARE MEASUREMENT

maintainability and efficiency [29], but also to estimating the implementation cost and effort to support different features. According to the measurement theory [30], measurement is a process in which numbers (or symbols) from the mathematical world are assigned to different entities from the empirical world to describe the entities according to defined rules. A measure represents a variable to which a value is assigned as a result of the measurement [31]. In the papers of this thesis, we also refer to measures as metrics [32].

The measurement theory describes how to construct measures (metrics) and formalize their mapping from the empirical to the mathematical world, i.e., empirical relations between entities are mapped to mathematical relations so that they can be analyzed. For example, if two software systems (A and B) are related by the empirical relation "more complex than", we can define a measure of their complexity ($c$), where measurement results are related by the mathematical relation "$>$". The mapping between the empirical and mathematical relations, therefore, is defined as:

$$c(A) > c(B) \Rightarrow A \ more \ complex \ than \ B$$  \hspace{1cm} (1.1)

Measurement results can be represented on different scales and different relations between the results are possible depending on the scale [30]:

- **Nominal scale** - only a relation of equivalence possible ($A = B$)
- **Ordinal scale** - a nominal relation + greater/smaller than ($A > B$)
- **Interval scale** - an ordinal relation + difference computation ($A - B$)
- **Ratio scale** - an interval relation + ratio computation ($A / B$)
- **Absolute scale** - like ratio, but counts the number of items

In order to structure the software measures according to their objectives, the Goal Question Metric (GQM) approach proposed by Basili et al. can be used [33]. The GQM defines the measurement as a mechanism that helps to answer a variety of questions about the software process and products. It defines the measurement model on three levels: Conceptual (goals), Operational (questions) and Quantitative level (metrics) as shown in Figure 1.7:

![Figure 1.7: The Goal Question Metric (GQM) levels [33]](image)

The goal is defined for one object (e.g., a product or process) with respect to its quality attributes for a specific purpose (e.g., an evaluation) and from a specific perspective (e.g., that of a system designer). A set of questions for one goal is used to specify how the goal will be assessed by characterizing the object to be measured. Finally, the measures represent the quantitative data associated with every question that enable the question to be answered.
CHAPTER 1. INTRODUCTION

1.3.1 Measurement process

The ISO/IEC 15939 standard for the measurement process in software engineering [34] defines the process as a set of activities that are required to specify: (i) what information is needed for the measurement, (ii) how the measures are made and measurement results are analyzed, and (iii) how the results are validated. Additionally, the measurement process specifies how to build the measurement products, although this area is beyond the scope of this thesis. A simplified measurement process is shown in Figure 1.8:

![Figure 1.8: Measurement process activities [34]](image)

Activity (1) defines the scope of the measurement and who will execute it. Activity (2) elaborates on the measurement plan, such as what is to be measured (i.e., which entities and their quality attributes), what information is needed (i.e., the reason for the measurement), which measures will be used and on what scale, and the criteria for evaluating the measurement results. Activity (3) describes how the data will be collected and analyzed. Finally, activity (4) describes how the measures and the measurement process will be evaluated based on the defined criteria of activity (3). This process is defined as iterative in order to improve both the measures and measurement process based on the results of the evaluation in activity (4).

One important segment of the planning activity (3) is to ensure that the measures are clearly defined in order to avoid different interpretations of how the measurement has been done (see the measurement errors of different implementations of the commonly known lines of code measure [35]). The measures will be defined based on the conceptual model, which is used to describe the entities in the empirical world [36] to ensure that the metrics can satisfy the required information need.

Software measures are usually defined using either set theory or algebra expressions. In order to prevent a definition of a measure using one of these two approaches from becoming too complex, however, alternative approaches can be taken, such as using pseudo-code snippets. For example, the complexity $(c)$ of one software component $(x)$ can be defined using algebra as follows:

$$c(x) = \sum_{i=1}^{n} r_i(x) * \sum_{i=1}^{n} t_i(x);$$

$$r_i(x) = \begin{cases} 1, & \text{if } x \text{ receives } sig_i \\ 0, & \text{otherwise} \end{cases}$$

$$t_i(x) = \begin{cases} 1, & \text{if } x \text{ transmits } sig_i \\ 0, & \text{otherwise} \end{cases}$$

where $n$ represents the total number of signals and $sig_i$ the signal with serial-number $i$. Using set theory, this same result can be achieved by defining two sets $Sin(x)$ and $Sout(x)$:
1.3. SOFTWARE MEASUREMENT

- $Sin(x) = \{sin_1(x), sin_2(x), ..., sin_\alpha(x)\}$ - a set of signals received by software component $x$.

- $Sout(x) = \{sout_1(x), sout_2(x), ..., sout_\beta(x)\}$ - a set signals transmitted by software component $x$.

The complexity would then be calculated as follows:

$$C(x) = |Sin(x)| \times |Sout(x)|$$

Finally, the corresponding pseudo-code could look like this:

```java
int Complexity(Component x)
{
    int Sin = 0;
    int Sout = 0;
    foreach (Signal in ReceivedSignals(x))
    {
        Sin = Sin + 1;
    }
    foreach (Signal in TransmittedSignals(x))
    {
        Sout = Sout + 1;
    }
    return Sin * Sout;
}
```

An important part of the measurement process is to validate the defined software measures. Two different types of validation can be performed [37]: a theoretical validation to answer "Are we measuring the right attribute?" and an empirical validation to answer "Is the measure useful?".

The theoretical validation ensures that a measure does not violate the properties of the measured entity [38]. This can be achieved by assessing whether the measure satisfies certain theoretical criteria. For example, there must be at least two entities for which the measure yields a different result, measuring the same entity twice yields the same results, measuring two entities can yield the same result, etc. Additionally, Briand et al. [39] classify the measures according to five attributes (size, length, complexity, coupling and cohesion) and define a set of properties required to measure each attribute. These properties can be used to group measures according to different properties and validate that they do indeed measure an intended attribute.

The empirical validation ensures that the measurement results are consistent with expected values [38]. This can be achieved by discussing the results with experts who work with the measured entity to ensure that they are consistent with their expectations (e.g., code complexity should decrease after re-factoring it). Statistical analysis techniques can also be used to validate the relationship between two attributes if the measurement goal was to explain one attribute which, for example, cannot be measured by measuring another attribute. An example of this is the use of a correlation analysis based on historical data to validate the link between code complexity and the number of faults.
1.3.2 The use of software measurement in this thesis

The results presented in this thesis rely heavily on the use of software measures and measurement results. The automotive architectural models and meta-models represent the scope of the measurements, and our main information need was to assess their evolution with respect to a number of properties, including changes, size, complexity and coupling. Since available, off-the-shelf measures do not use the specific modeling characteristics of the automotive domain, we defined our own set of measures for this purpose.

The measures are defined following the GQM approach based on the appropriate conceptual models. Since we performed several studies to address different goals and research questions, we followed the GQM approach because it helped us to structure the measures in order to address all of the goals and questions. For example, in Paper B, we defined a measure of meta-model change in order to analyze the evolution of domain-specific meta-models, whilst in Paper C, we defined a measure of meta-model size, length, complexity, coupling and cohesion in order to assess the impact of the meta-model evolution on different roles in the development process.

Clearly defining both the goals and questions for each study also enabled us to re-use the logic behind certain measures used for measuring different entities in order to address similar questions and goals about these entities. For example, the goal of our complexity and coupling measures defined in Paper A was to monitor the evolution of architectural models and was based on the research question: What is the trend in the complexity and coupling evolution of these models? One of the goals of Paper C was to monitor the evolution of the same properties for the domain-specific meta-models based on the research question: What is the impact of meta-model evolution on different roles in the development process?

Consequently, we re-used the logic behind our complexity and coupling measures for software components of the architectural models (the definition of which was based on the conceptual model presented in Paper A) and redefined the measures so that they would apply to meta-classes of the domain-specific meta-models (based on the conceptual model presented in Paper C). In particular, the measures defined in Paper A were based on the interaction between different software components using signals, whilst those of Paper C were based on the interaction between different meta-classes using associations.

Some of our measurement results are presented on the ratio scale (e.g., model and meta-model complexity) whilst others are shown on the absolute scale (e.g., the number of meta-model changes). The majority of measures are defined using either set theory or algebra. We defined the measure of the complexity and coupling of the architectural models in Paper A using algebra, and the measure of size, length, complexity, coupling and cohesion of the meta-models in Paper C using set theory. Data collection was automatic because we had developed tools to measure the properties of both the models and meta-models based on the appropriate conceptual model.

The measures used in our studies are validated both theoretically and empirically. The theoretical validation was based on the properties defined by Briand et al. [39] for the size, length, complexity, coupling or cohesion measures depending on the classification of the measures. The empirical validation
was based on applying the measures on project data from Volvo Cars and the AUTOSAR consortium.

The complexity and coupling measures for monitoring the evolution of the architectural models were applied to a number of software components and ECUs from two evolving electrical systems at Volvo Cars. The measurement results were presented to experts who were able to confirm the results based on their knowledge about actual system changes. The measure of meta-model change was applied to a number of AUTOSAR meta-model releases, and the results of the measurements matched the change documentation of AUTOSAR. Finally, the measures for monitoring the size, length, complexity, coupling and cohesion of domain-specific meta-models were applied to a number of historical releases of the AUTOSAR meta-model and the choice of the most suitable measures was based on statistical analysis (correlation and principal component analysis).

A summary of the measures used in our studies is shown in Table 1.1.

<table>
<thead>
<tr>
<th>Measure name</th>
<th>Measure goal</th>
<th>Defined</th>
<th>Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complexity metric</td>
<td>Monitoring the complexity evolution of the automotive software systems</td>
<td>Paper A</td>
<td>Paper A</td>
</tr>
<tr>
<td>Package coupling metric (SW components)</td>
<td>Monitoring the coupling evolution of the automotive software systems</td>
<td>Paper A</td>
<td>Paper A</td>
</tr>
<tr>
<td>Number of changes (including Number of changed elements and attributes)</td>
<td>Estimating the effort / cost of adopting a new meta-model version / feature and finding the optimal set of features to be adopted</td>
<td>Paper B</td>
<td>Papers B, D, E (textual description)</td>
</tr>
<tr>
<td>Number of classes and Number of attributes</td>
<td>Monitoring the size of meta-model evolution</td>
<td>Paper C</td>
<td>Papers B, C, E</td>
</tr>
<tr>
<td>Average depth of inheritance</td>
<td>Monitoring the length of meta-model evolution</td>
<td>Paper C</td>
<td>Papers B, C, E</td>
</tr>
<tr>
<td>Fan-in, Fan-out and Fan-in-out</td>
<td>Monitoring the complexity of meta-model evolution</td>
<td>Paper C</td>
<td>Papers C, E</td>
</tr>
<tr>
<td>Package coupling metric (classes) and Coupling between objects</td>
<td>Monitoring the coupling of meta-model evolution</td>
<td>Paper C</td>
<td>Paper C, E</td>
</tr>
<tr>
<td>Package cohesion metric and Cohesion ration</td>
<td>Monitoring the cohesion of meta-model evolution</td>
<td>Paper C</td>
<td>Papers C, E</td>
</tr>
</tbody>
</table>

1.4 Research questions and contributions

The main objective of this thesis was to enable faster adoption of new architectural features in automotive software development projects. Therefore, we defined the following, key research question:
Q How can the evolution of architectural models and meta-models related to a set of new architectural features be managed efficiently?

In order to answer our key research question, we divided it into several, smaller research questions that we addressed in different studies. As a motivational study, we analyzed the evolution of architectural models of automotive software systems in order to understand the complexity impact of adding new car functionalities to the systems. Moreover, our objective for this first step was to define a set of software measures capable of quantifying the complexity and coupling properties of the architectural models. Thus, we defined the following research questions:

Q1 How can a change in the complexity and coupling of automotive architectural models be measured?

Q2 What is the trend in the complexity evolution of architectural models during the lifetime of one automotive software system?

The results showed that the changes in the models were significant and a deeper understanding of the source of these changes was required. Our next step, therefore, was to analyze the changes between different versions of the automotive domain-specific meta-model, which specifies how different architectural features are modeled. This analysis was facilitated by defining the following research questions:

Q3 How can a measure of change between different domain-specific meta-model versions be defined?

Q4 Can this measure of change be used to monitor the evolution of domain-specific meta-models?

The results showed that different meta-model changes usually impact the models and tools developed by different roles in the automotive software development process, so the meta-model changes for each role were analyzed separately. For this reason, we also defined the following research questions:

Q5 What major roles exist in the automotive software development process?

Q6 Which parts of the domain-specific meta-model are relevant for the modeling tools used by these roles?

Q7 Which roles are affected most by the evolution of the domain-specific meta-model?

The results showed that monitoring only the number of changes may not suffice for an accurate impact assessment of the new, domain-specific meta-model versions and their features on different roles. This is because some changes only affect certain roles, whilst others affect multiple roles. The latter changes are considered to be more severe, such that monitoring other meta-model properties, for example, complexity, coupling and cohesion, is also important. This led us to define the following research questions:
Q8 Which measures are the most suitable to measure the evolution of domain-specific meta-models with respect to their size, length, complexity, coupling and cohesion properties?

Q9 How can the results of different software measures be combined in order to assess the impact of meta-model changes on different roles?

Up until this point, the analysis showed that not all architectural features from the new domain-specific meta-model versions are relevant for all roles involved in the development process. This means that a subset of features is relevant to a subset of roles, such that the roles usually must decide which new architectural features to adopt in the development projects. Therefore, as it is also important to analyze the changes in new meta-model versions related to a subset of their features, we defined the following research questions:

Q10 How can the evolution of domain-specific meta-models with respect to their different architectural features be quantified?

Q11 How can an optimal set of new architectural features that are to be adopted in the software development projects be selected efficiently?

We combined the answers to research questions Q1-Q11 in order to answer our main research question Q. Specifically, we used (i) the identified roles, (ii) the defined software measures, and (iii) the method for selecting an optimal set of architectural features that are to be adopted in the development project in order to perform the role-based analysis of domain-specific meta-model evolution related to its architectural features.

1.4.1 Paper contributions

All research questions (Q1-Q11) are addressed in one of the papers of this thesis. Table 1.2 shows which questions were addressed in which paper.

<table>
<thead>
<tr>
<th>Paper</th>
<th>Addressed research question</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paper A</td>
<td>Q1, Q2</td>
</tr>
<tr>
<td>Paper B</td>
<td>Q3, Q4, Q5, Q6, Q7</td>
</tr>
<tr>
<td>Paper C</td>
<td>Q8, Q9</td>
</tr>
<tr>
<td>Paper D</td>
<td>Q10, Q11</td>
</tr>
</tbody>
</table>

In Paper A, we concluded that the architecture of automotive software systems is very complex; this complexity is constantly increasing with the addition of new car functionalities. Therefore, measuring the increase in complexity and coupling of these architectures is an important aspect in reducing faults and increasing system longevity. The measures proposed in this paper can be used for this purpose. To automate both the measurement process and presentation of measurement results, we developed a tool (QTool), which can be used during the evolution of automotive software systems in order to indicate when and where a set of architectural changes can be performed to reduce system complexity.
In Paper B, we concluded that a role-based analysis of changes between different domain-specific meta-model versions is valuable in estimating the effort needed to adopt certain meta-model versions in development projects. We defined a method that can successfully monitor the evolution of domain-specific meta-models (i.e., the changes) for different roles in the development process. We also identified the major roles in the automotive development process that are most affected by changes in the domain-specific meta-model used in the automotive industry.

In Paper C, we assessed a number of software measures that are applicable for measuring the evolution of domain-specific meta-models with respect to five properties: size, length, complexity, coupling and cohesion. We also showed how to combine the results of these measures in order to assess the impact of adopting new version of the domain-specific meta-models on the modelling tools used by different roles in the development process.

In Paper D, we defined a method (MeFiA - Meta-model Feature Impact Assessment) that can identify an optimal set of new architectural features that are to be adopted in the development projects. The method is based on the impact of these features on domain-specific meta-models used in the development and feature prioritization. In particular, we identify which meta-model changes are related to which architectural feature.

In addition to papers A-D, which answer the research questions Q1-Q11, Paper E has also been included in this thesis. Paper E presents a tool (ARCA) that can automatically perform the analysis described in papers B, C and D, thereby answering research questions Q3-Q11. The ARCA tool can be used to facilitate the evolution of automotive software architectures related to the adoption of new architectural features, as defined in the main research question Q. In particular, the tool can assess the impact of meta-model changes on modeling tools used by different roles. This impact is used in the analysis of which new meta-model versions or a subset of their features will be adopted in the development process of different projects.

1.4.2 Industrial contributions

All results presented in this thesis are directly deployed at Volvo Cars by means of incorporating the two implemented tools QTool and ARCA in the development process of automotive software systems.

The QTool implements the complexity and coupling measures presented in Paper A and is primarily used by the automotive Software Architecture Testers. The tool is used during the evolution of an automotive software system after the addition of new functionalities in order to analyze their impact on the complexity and coupling properties of different architectural components (e.g., sub-systems, ECUs and domains). If the results are unsatisfactory, the architectural components must be re-designed so as to reduce coupling and increase the ECU cohesion. This could be done by re-allocating a subset of the software components (i.e., functionalities) onto other ECUs, which might immediately result in a reduced number of signals on the electronic buses.

The ARCA tool described in Paper E implements both the measures and methods that are defined in papers B-D, and is used by Automotive System Designers. There are three, main situations in which this tool can be used:
1. To analyze changes between AUTOSAR meta-model releases

The analysis is done by four different teams at Volvo Cars for four main roles in the AUTOSAR-based software development process. The System Architects (i) and System Designers (ii) analyze the impact of the changes on the Application Software Designers role. The Signal Database Team (iii) analyzes the impact of the changes on the ECU Communication Designers role. Finally, the AUTOSAR Team (iv) analyzes the impact of the changes on the ECU Communication Configurators and ECU Diagnostic Configurators roles, as well as on other parts of the ECU configuration, such as non-volatile memory and encryption library.

The aim of this analysis is to facilitate the decisions concerning which new AUTOSAR release will be adopted in the development process as part of a cost-benefit analysis (Volvo Cars currently uses AUTOSAR release 4.0.3 and is analyzing the impact of switching to one of the 4.1.1-4.2.1 AUTOSAR releases). The ARCA tool can be used to estimate the effort required to implement the changes for each analyzed role and will be complemented by an analysis of the actual need for adopting new AUTOSAR releases.

2. To analyze changes related to specific AUTOSAR features

The analysis is done primarily by the System Architects at Volvo Cars with support from the System Designers and AUTOSAR Team. It includes the impact assessment of adopting each new AUTOSAR feature on the four main roles in the AUTOSAR based software development (Application Software Designers, ECU Communication Designers, ECU Diagnostic Configurators and ECU Communication Configurators) and identification of the optimal set of features to be adopted using the MeFiA method.

The aim of this analysis is to facilitate the decisions concerning which new AUTOSAR features will be supported as part of a cost-benefit analysis (Volvo Cars currently analyzes the impact of adopting new features from the AUTOSAR release 4.2.1). The ARCA tool can be used to estimate the effort needed to implement the changes for the analyzed feature and will be complemented by an analysis of the actual need for adopting this feature.

3. To analyze changes between SVN versions of the AUTOSAR meta-model during the development of one release

This analysis is done by the AUTOSAR Team at Volvo Cars with the aim of continuously following-up changes in the AUTOSAR meta-model in order to have sufficient time to influence their standardization (the tool is currently used to analyze changes for the AUTOSAR release 4.2.2).

For example, if an accepted change in the AUTOSAR meta-model removes one meta-element that is widely used at Volvo Cars, the change will be discussed again in the AUTOSAR consortium involving representatives from Volvo Cars before its official release. This is particularly important for companies that are not AUTOSAR core partners (companies driving the AUTOSAR standard) because they tend not to be represented in all AUTOSAR groups, making it more difficult for them to be aware of all of the changes. Additionally, core partners have a certain period of time (i.e., a few weeks)
before the official release to review and block any critical changes. This is not, however, the case with other AUTOSAR partners, who must invest additional effort to influence the changes during the implementation phase of one release. In such a situation, the ARCA tool could be a valuable aid.

As well as the implementation of these thesis results at Volvo Cars, the ARCA tool functionality related to a comparison of different SVN versions of the AUTOSAR meta-model for a particular new feature is planned to be adopted in the daily change management process of AUTOSAR. The idea is to present the actual changes in the AUTOSAR meta-model in corresponding Bugzilla\textsuperscript{1} implementation tasks so that the reviewer can confirm that all changes are really intended. This information will also be used to generate the change documentation between different AUTOSAR releases and traceability of the AUTOSAR meta-model changes to the Bugzilla implementation tasks.

1.5 Research methodology

Research methodology describes the systematic process of data collection and analysis in order to achieve the desired conclusions. The general research methodology used in the studies of this thesis is action research. Using only one research method, however, does not usually suffice for the combined work of experts and researchers [40]. Therefore, we reported each cycle of the action research as one case study.

In this section, we summarize the theory of the action research and the case study, demonstrate how we described each action research cycle as one case study and finally, discuss the validity of our results.

1.5.1 Theory of action research

The term ”action research” was first introduced by Kurt Lewin in 1946 [41] as a research approach in social sciences intended to improve inter-group relations. Lewin acknowledged the necessity of the existing scientific approaches of acquiring general knowledge (e.g., by conducting surveys) and situation specific knowledge (e.g., by conducting experiments), but questioned their value if they had no practical use (”Research that produces nothing but books is not sufficient.”). Therefore, he called for a research type that immediately leads to social actions in order to help experts deal with the actual problems. One definition of action research commonly used was designed by Rapoport [42]:

”Action research aims to contribute both to the practical concerns of people in an immediate problematic situation and to the goals of social science by joint collaboration within a mutually acceptable ethical framework.”

Therefore, the main aim of action research is to improve the practice and increase collaboration between the experts and scientists [43]. This is further elaborated by Susman et al. [44] who define a circle of five steps around an evolving client system (e.g., a company facing practical problems) that will be conducted during the action research process, as shown in Figure 1.9.

\textsuperscript{1}A tool used by AUTOSAR for issuing and documenting change requests.
The diagnosing phase focuses on identifying the practical work problems faced by the experts from companies involved in the action research project. The next phase involves the initial planning of actions that must be taken in order to solve the identified problems. These actions are usually undertaken by the corporate experts, although action research can also be involved. The evaluation phase assesses the result of applying the actions at the companies by analyzing if the identified problems still exist. If not all defined problems are solved, the entire process is repeated taking into consideration inputs from the previous phases in re-defining the actions to be taken. The last phase of the cycle is driven by the action researcher and is concerned with identifying overall findings that are important for the generalization of the results to create scientific knowledge applicable to other, similar cases (e.g., at other companies facing similar problems).

The action researcher does not have to be involved in all five phases. Depending on the involvement of the action researcher in different phases, we can distinguish between the following main types of action research [45]:

- **Diagnostic** - action researcher is involved in diagnosing the problem
- **Empirical** - action researcher is involved in evaluating the results
- **Participant** - action researcher is involved in diagnosing and planning
- **Experimental** - action researcher is involved in nearly all phases

### 1.5.2 Theory of case study

A case study is classified as an empirical research method [46] and focuses on the examination of a real-world situation making it quite suitable for industrial evaluations. Yin [47] defines a case study as an iterative process consisting of five phases, as shown in Figure Figure 1.10:
Here, we focus on the design, collect and analyze phases. The design phase consists of the following five components [47]: (i) Research questions, (ii) Study propositions, (iii) Unit of analysis, (iv) Linking data to prepositions and (v) Criteria for interpreting the findings.

The design of each study begins with a clear definition of the research questions. Providing answers to these questions is the main objective of a case study. In order to understand how to achieve this goal, however, the scope of the study is defined together with the identification of elements that are to be examined (the study propositions). Next, the unit of analysis (i.e., the ”case”), the elements of which are to be examined, is defined. The data obtained from this examination are then linked to the propositions in order to answer the defined research questions using, for example, statistical analysis. Finally, the criteria for data interpretation are defined in order to indicate when the obtained results can be considered valid, for example, by defining the statistical significance if statistical methods are used to analyze data.

The data collection phase can include both qualitative and quantitative data collection methods [46]. Qualitative data can be obtained by analyzing documentation, performing observations, conducting interviews, etc. Interviews are especially common in analyzing industrial cases because they provide quick answers to question from experts. They can be formal with a precisely-defined set of questions, informal relying on a casual discussion with experts [48] or semi-formal where questions are pre-defined, but can be deviated from during the interview [49]. Quantitative research represents the analysis of numerical data in order to explain a certain phenomenon [50]; quantitative data are usually obtained by measurement.

Data can also be analyzed using different methods, such as pattern matching (comparing the empirical pattern with one or several predicted patterns) and explanation building (e.g., using theoretically proven concepts) [47]. Quantitative data can be analyzed using a number of statistical methods, including correlation analysis and time-series analysis.
1.5.3 One action research cycle as a Case study

Each action research cycle can be seen as one case study that aims to answer a set of research questions related to the study context relevant for this cycle. Understanding this context is crucial for defining the actions that can solve the identified problems. Additionally, the case study analysis provides an immediate possibility to evaluate the results of the actions on the chosen unit of analysis. On the other hand, action research methodology provides the possibility to collate the results of different case studies into one method (i.e., a set of actions) that can address the general problem. Action research also emphasizes the need to discuss how to generalize the proposed method, thereby raising the level of its scientific significance.

The definition of research questions is part of the diagnosing phase of the action research. The choice of the unit of analysis, the study propositions and the description of how to obtain the data and link them to the chosen propositions (e.g., a method definition) are all part of the action planning phase. The action-taking phase includes the application of the proposed method on the unit of analysis and data collection. Finally, data evaluation corresponds to the data analysis phase of the case study with focus on the validation of the performed actions (e.g., the proposed method). The fact that each new case study may define new research questions and use different units of analysis is important for advancing the knowledge between different action research cycles because the results of one cycle can be used as input for the next cycle.

In the studies included in this thesis, we conducted an experimental action research project at Volvo Cars where the author of the thesis was the action researcher working at the company. The general aim of the project was to develop methods and tools to address the main research question \( Q \), which was divided into smaller research questions \( Q1-Q11 \) as explained in Section 1.4. We conducted four action research cycles, each based on one case study with the aim of addressing one or several smaller research questions. Each case study was documented in one of the thesis papers. The first case study served as a motivational study and the results of the second case study served as input to the subsequent case studies.

Case study 1 (research questions \( Q1 \) and \( Q2 \))

The action researcher was part of the Software Architecture Testing Team at Volvo Cars. The diagnosing phase was performed at the start when the problem of how to monitor the complexity of the automotive architectural models was defined. In the planning phase, it was decided that the architectural models used at Volvo Cars would be the unit of analysis. The outcome of the case study was a method (based on two measures) and a tool for monitoring the complexity of the automotive architectural models. The method was validated based on its application on two evolving architectures and the tool was included in the software architecture testing process at Volvo Cars.

Case study 2 (research questions \( Q3, Q4, Q5, Q6 \) and \( Q7 \))

The action researcher was part of the AUTOSAR Team at Volvo Cars. The diagnosing phase was performed at the start when the problem of how to monitor the evolution of domain-specific meta-models for different roles involved
in the development was defined. In the planning phase, it was decided that the AUTOSAR meta-model would be the unit of analysis. The main outcome of this case study was a method (based on a measure of change) for monitoring the evolution of domain-specific meta-models for different roles. The method was validated based on its application on a number of AUTOSAR meta-model releases for seven main roles in the automotive software development process. The roles were identified using semi-structured interviews with a number of engineers from automotive OEM and software supplier companies.

Case study 3 (research questions Q8 and Q9)

The action researcher was part of the AUTOSAR Team at Volvo Cars. The diagnosing phase was performed at the start when the problem of which software measures would be used to assess the impact of changes in domain-specific meta-models on different roles was defined. In the planning phase, it was decided that the AUTOSAR meta-model would be the unit of analysis. The identified roles and defined data model in the Case Study 1 served as an input to this analysis. The main outcome of Case Study 3 was a method (based on a combination of measures) for assessing the impact of changes in domain-specific meta-models on different roles. The method was validated based on its application to historical releases of the AUTOSAR meta-model. Correlation and principal component analyses were used to identify the most suitable measures.

Case study 4 (research questions Q10 and Q11)

The action researcher was part of the AUTOSAR Team at Volvo Cars. The diagnosing phase was performed at the start where the problem of how to identify an optimal set of new architectural features to be adopted in the development projects was defined. In the planning phase, it was decided that the AUTOSAR features would be the unit of analysis. The identified roles and defined measure of change in Case Study 1 served as an input to this analysis. The main outcome of this case study was a method (based on the measure of change) for identifying an optimal set of architectural features to be adopted in development projects. The method was validated based on its application on 14 new features of the AUTOSAR release 4.2.1.

Based on the methods defined in case studies 2, 3 and 4, we developed a tool presented in Paper E. The tool was included in the process of analyzing the impact of new AUTOSAR meta-model releases and their features on different roles involved in software development at Volvo Cars.

Since AUTOSAR and the AUTOSAR meta-model were used as a unit of analysis in case studies 2, 3 and 4, the action researcher was appointed to be one of the representatives of Volvo Cars in the AUTOSAR consortium. The aim was to ensure direct contact between the action researcher and experts from the AUTOSAR consortium. This in turn enabled fast feedback loops in the action planning and result evaluation phases by a number of experts in the field, including OEMs and different types of suppliers.
1.5.4 Research validity

We followed the principles of Baskerville et al. [51] to increase rigor during the action research project, with a particular focus on:

- Maintaining collaboration between the action researcher and experts during all phases of the action research.
- Promoting iterations of different phases, particularly action planning, execution and evaluation of the results.
- Ensuring generalization of results in the specifying learning phase.

According to Cook and Campbell [52], four types of validity threats to empirical studies conducted in the area of software engineering must be considered. We explain how we addressed each of these in our four case studies (four action research cycles) below.

**Internal validity**

Internal validity is concerned with the results of the analysis not being casual, i.e., the relationship between the measured properties and the outcome should not be random. The most severe threat to the internal validity in our studies was related to the measurement process which was performed by developing two software tools for calculating the measures. In order to ensure internal validity, we performed detailed testing of the tools using smaller examples before employing them for the main measurements.

**External validity**

External validity concerns generalization of results and is one of the most prominent threats to action research validity (i.e., the applicability of the results to other companies facing similar problems) [51]. The reasons for this involve a deep involvement of both the experts and action researcher in the working practice of the company undergoing analysis and the evaluation of the proposed methods by applying them to this specific context.

There were two particular threats to the external validity in our studies. The first was that the proposed methods and tools would apply only to the automotive software development process at Volvo Cars and not to other automotive companies. Therefore, we included several other companies (both OEMs and suppliers) in the evaluation of the proposed methods and tools.

The second threat was related to the AUTOSAR meta-model that was the unit of analysis in case studies 2, 3 and 4. The proposed methods and tools that we applied to the AUTOSAR meta-model should also be applicable to other domain-specific meta-models. Therefore, we mapped the layers of the AUTOSAR modelling environment to the layers of MOF that are commonly accepted layers for modelling in different domains. In addition, we discussed the steps that must be taken in order to apply the proposed methods to meta-models of other domains, such as avionics, telecommunications and banking, in the thesis papers.
Construct validity

Construct validity concerns the mismatch between the theory and observations. In our studies, this was related to the ability of the measures to capture the desired system properties. We ensured this by the theoretical validation of measures presented in Case Study 1 and because the measures used in Case Study 3 were based on commonly-used UML measures previously proven in the scientific world. Finally, the measure of change used in case studies 2 and 4 was defined according to the GQM approach based on the conceptual model, which ensured that all relevant meta-model changes were captured.

Conclusion validity

Conclusion validity concerns the degree to which the conclusions of the studies are reasonable. In Case Study 3, this was related to the significance of the results obtained by the statistical analysis, which was high. In our other case studies, the conclusions were derived based on applying the methods to industrial scenarios and comparing the results with the expectations of experts. The conclusion was that the results could capture the desired properties.

1.6 Future work

Our future work will follow two major directions. Firstly, we plan to apply the methods and tools proposed in this thesis to the analysis of domain-specific meta-model evolution on meta-models from other, non-automotive industries. If these domain-specific meta-models are not available to us, we will apply the methods to the evolution of the UML meta-model.

Secondly, we plan to extend our analysis of the software model and meta-model evolution to other artifacts in the automotive development process, such as system requirements. In particular, we plan to analyze the co-evolution of the software models, meta-models and requirements in the automotive domain and answer research questions, such as: "How can the architectural models and system requirements be evolved efficiently when adopting new meta-model versions in automotive development projects?".

1.7 Personal contribution

The author of this thesis has been the main contributor as regards the planning and execution of the studies described in the thesis, and writing of the included publications.
Bibliography


Chapter 2

Paper A

Measuring the Impact of Changes to the Complexity and Coupling Properties of Automotive Software Systems

D. Durisic, M. Nilsson, M. Staron and J. Hansson

Abstract

**Background:** In the past few decades, exponential increase in the amount of software used in cars has been recorded together with enhanced requirements for functional safety of their embedded software. As the evolution of electrical systems in cars often entails changes to automotive software architecture, it is important to monitor the impact of these changes on its quality attributes.

**Method:** We conducted a case study analysis of a distributed electrical system developed at Volvo Car Corporation and deployed to Volvo Cars. The goal was to develop, apply and evaluate measures of complexity and coupling which could support software architects in monitoring the architectural changes.

**Results:** The results showed that two metrics - structural complexity and coupling - can guide architectural work and turn the attention of architects to most complex subsystems. The results were confirmed by monitoring a complete electrical system of a vehicle under two releases.

**Conclusion:** By applying the metrics after each major change in the architecture, it is possible to verify that certain quality attributes have not deteriorated and to identify new testing areas. Using these metrics increases the product quality with respect to stability, reliability and maintainability and also has a potential to reduce long-term software development/maintenance costs.
2.1 Introduction

The amount of software in today’s cars has reached one gigabyte of on-board binary code (excluding the infotainment), and is constantly increasing [1]. The increase is explained by the fact that more than 80% of innovations in cars are related to software and the majority of them increase the interaction between previously less dependent parts of the system [2]. A good example of this is the “pedestrian detection” technology which aims to prevent a significant number of pedestrian-involved accidents [3].

At the same time as the amount of software-realized functionality in cars increases, quality demands for safety, reliability and performance shall remain high for the entire car product, including software. Most of the quality attributes are actually improved with the use of software. However, huge binary code increases the probability of fault propagation in already complex software systems (with up to 2000 software-based functions where more than 10% involve user interaction, e.g., setting the temperature), resulting in significantly harder integration testing.

In the light of these dynamic changes in the automotive development, two articles can be cited that show the importance of software architecture in automotive software systems and the process of its evolution: one by Eklund and Olsson [4] studying the factors influencing the architecture using Architecture Business Cycle model [5], and one by Axelsson [6] showing the evolution of architecture in different product lines.

Although evolution of software architectures is a natural phenomenon, it has a new dimension in the automotive software. Automotive software can be considered as mixed-criticality system since it contains functions that control safety-critical behavior of the car and user functions often in the same execution space and/or hardware. Therefore architectural changes are important to be monitored in order to identify potential risks to safety-critical functions which may have been caused by non-safety-critical functions. An example of architectural changes can be seen in the evolution of car’s headlights. The initial software controlling this unit in a car was implemented just to turn the lights on and off. The second version was able to manually adjust the beam of light and turn it along the vertical axis. Finally the current version is able to turn the lights in both directions, horizontally and vertically, automatically following the car’s movement in curves or when crossing a speed bump.

Even in case of their high architectural significance, it is usually very inefficient to wait for a new platform release to implement the changes since platform’s lifecycle is quite long today. Due to the high number of competitors on the market, product quality is vital but not sufficient to sell the expensive product. For this reason, and implied by the low production cost demand, one system platform should be designed to endure all changes and have a satisfying quality for at least 5-6 years. Under these circumstances, the platform’s maintainability properties and the change management process play one of the most important roles.

---

1Platform represents a complete electrical system used for a particular product line (a set of car models). It is a distributed system with typically 70-100 ECUs (Electronic Control Units) each representing one embedded computer system (hardware with software) in charge of one or more electrical systems in the platform, e.g., engine control, breaking, radio, etc.
Apart from their frequency implying the risk of deteriorated quality, software changes in automotive systems can cause two additional problems:

1. Higher integration and regression costs: research shows that only 25% of functionalities are created inside car companies (Original Equipment Manufacturers - OEMs), while the rest is just integrated after the delivery from suppliers [2]. This approach increases the quality of delivered components since the suppliers get quite experienced while delivering similar components to different OEMs. However, it also increases the development cost since it most often requires modifications and upgrades of already implemented components. Such a distributed development makes communication between OEMs and suppliers extremely difficult, especially during the development process.

2. Most of the changes are architectural: evolution of software functions affects the architecture of automotive software systems as most of the changes are either additions or improvements of the existing functionalities represented with new signals on the electronic buses [7]. Architectural changes are more likely to cause scattering of code through different sub-systems potentially causing serious malfunctions in others [8]. For example during integration testing, it may happen that a tester is, for no specific reason, unable to start the engine of a car even though is starts normally just a few seconds later. The explanation for this behavior most probably lies in the start-up process which initiates many checks and at least one of them fails. The reason for this could be the existence of an error in one of the sub-systems which may not be related to the engine, gear or other important start-up modules. Still, due to the high interaction between sub-systems, the error is able to propagate and create an incorrect state resulting in abortion of the car’s start-up process. This phenomenon when a relatively small fault in one part of the system manifests as a serious malfunction in another is known as ”ripple effect” [9] and it represents one of the biggest threats to software systems which is significantly increased with the introduction of architectural changes.

For these two reasons, foreseeing the possible impact of changes before their realization could be crucial for assuring the desired quality of the system. This way, potentially dangerous architectural decision can be re-analyzed very early which could reduce the number of late changes and therefore lower the production cost. Additionally, identifying the parts of the system affected by the most severe changes is a good indicator for the system testers where to put additional effort in order to eliminate potential ”ripple effects”.

Several metrics which are able to provide useful results based on the structural system requirements can be applied before sending change requests to suppliers. In this paper, we present two of the most applicable metrics for quantifying model complexity in embedded automotive software systems one based on modules complexity and the other based on modules coupling - since these metrics are directly related to the increase in the number of failures [10]. We also suggest how to interpret the measurement results in order to come to the conclusions which could imply future steps towards securing the desired quality attributes of the analyzed system - more precisely maintainability, reliability and robustness.
Since our metrics are meant to be applied in the early stages of the development process (i.e., before sending change requests to suppliers) where not many behavioral properties of the system are known, the metrics are mainly focused on structural system properties. One of such properties is inter-module communication — a property which increases the complexity of the system and impacts contracts between OEMs and suppliers. The metrics can identify early which parts of the system are mostly affected by the changes and allow OEMs to make necessary preparations which can significantly reduce the production cost. This paper presents a continuation of the work published in [11] by applying the metrics on the industrial system.

The rest of the paper is organized as follows: Section 2.2 describes the related work; Section 2.3 describes our research method; Section 2.4 describes the organization of the studied automotive software system at Volvo Car Corporation (VCC); Section 2.5 describes existing quality metrics and their adjustment to measure the complexity and coupling of automotive software systems; Section 2.6 describes the suggested way to present measurement results and methods for their interpretation; Section 2.7 describes an example of the automotive software system and demonstrates the use of presented metrics; Section 2.8 describes the theoretical and empirical validation of the metrics; finally, Section 2.9 describes the conclusions and discusses the future work.

2.2 Related Work

Several methods for measuring the size of architectural changes in software systems exist today. One of the most interesting methods is described in [12] with the goal to measure the distance between architectures over a certain number of releases based on the change in the structural connectivity of the architectural units. It can be complemented by other studies of change impact analysis on the architectural level based on the dependencies between architectural units, such as [13] and [14].

With regard to the system complexity, there are many different metrics used to measure the complexity, coupling and cohesion in software systems. Generally cohesion metrics are based on intra-module relations, coupling metrics are based on inter-module relations and complexity metrics can be based on either intra-module relations, inter-module relations (structural complexity), or both [15]. Since this paper observes automotive software systems from the perspective of OEMs where majority of modules are developed by the suppliers and delivered to OEMs as a "black box" platform specific executable code, it is not possible to apply most of the cohesion and complexity metrics based on intra-module properties available today. The reason for this is the fact that they are based on the analysis of source code (such as lines of code, the number of operators and operands [16], control graphs [17], syntactic constructs [18], or high abstraction level models [15]) which is usually not available to OEMs.

However, the information about the modules and their communication interfaces is available very early (on a design level done by OEMs) and that is why we based our metrics mostly on these structural properties. An alternative approach to this could be the use of FPA (Function Point Analysis) [19],
where each function is realized by one or more modules in the system. Then, the complexity of one module can be calculated as the sum of complexities of all of its realized functions.

The original complexity measure based on the strength of module dependencies is introduced by Stevens et al. [20]. Although there are a number of metrics based on this article, such as [21], [22], [23] and [24] or [25], many of them rely on the data obtained from source code (such as the number of input-output (IO) variables and methods invoked). As already mentioned in the context of automotive software design, the source code of components is usually not available to OEMs as it is developed by suppliers. Other metrics focus strictly on the dependencies between modules and the information exchange between them - denoted as structural metrics [16]. An accepted structural metric is the one based on modules fan-in and fan-out introduced by Henry et al. [26], but there are other metrics as well such as the Package Coupling Metrics (PCM) named and defined by Gupta et al. [27]. Due to the fact that cohesion calculations require knowledge about internal module properties usually captured in Simulink models and source code which were not available to us, we decided to exclude the cohesion from our research.

In order to choose the right metrics to measure the complexity and coupling of software components, sub-systems and ECUs in the system, we organized a workshop with the practitioners from VCC and presented them with the metrics mentioned above in order to discuss their applicability. In the workshop, we came to the conclusions presented in Table 2.1 for each one of them:

After discussing the applicability of these metrics, we concluded that none of them is fully applicable to the automotive domain. However, we also concluded that the Henry and Kafura’s Structure Complexity metric and the Package Coupling Metrics could be used as a basis for defining the applicable complexity and coupling metrics so we based our research on modifying them in order to fulfill the needs of the automotive electrical systems.

Despite the fact that there are a lot of books and papers available related to the complexity and coupling of software systems, we were unable to find many of these related to the automotive domain where dependability requirements of the distributed real-time software-hardware architectures require dedicated approach. Most of the things we found in the automotive domain were related to the AUTOSAR\(^2\) [28] and the principle of complex function decomposition using different software components. We also found many different tools available to support the design, implementation and testing of software components delivered by suppliers following the AUTOSAR standard, but we found no concrete measures for calculating the complexity and coupling between these components and/or between higher architectural units in the system in different system releases.

\section*{2.3 Research Method}

The formal definition of our research goal is defined according to the structure defined in [29] and [30] as:

\(^2\)AUTomotive Open System Architecture is a standard developed by OEMs, suppliers and tool developers in order to improve the development process and system quality.
Analyze the automotive electrical system for the purpose of measuring the effect of changes to its architectural properties, with respect to complexity and coupling, from the point of view of the system architects, designers and testers and in the context of the software systems developed at Volvo Car Corporation.

The research goal is driven by the need to assure maintainability, robustness and reliability quality attributes of the system during its architectural evolution and reduce the development/maintenance cost. In this paper, we show that this is tightly related to keeping the complexity and coupling increase during the architectural evolution of an automotive electrical system low and putting additional effort in testing parts of the system which suffered the highest complexity/coupling increase after the incorporation of the changes.

The research is conducted using the empirical research method [31, 32] based on the quantitative approach [33]. We first studied the development process of the automotive software systems at VCC with the aim to iden-
tify cause-effect relationships between the risk of deteriorated quality and the architectural changes. Our hypothesis was based on the assumption that measuring the size and impact of architectural changes before their realization by the suppliers can indicate early potentially bad architectural and design decisions. These decisions could affect the quality of the product and thus increase the production cost. By identifying these threats before the involvement of the suppliers, necessary measures can be taken promptly thus minimizing the costs.

After defining the research goal and hypothesis, we conducted a case study analysis [34] on several different metrics in the software system in cars in order to test their applicability to automotive domain. We concluded, together with our industrial partners from Volvo, that the metrics based on the structural complexity and coupling changes in the system are the most suitable metrics. One of the main reasons for focusing on the structural metrics based on architectural design is the necessity to apply them early. In addition, since the existing metrics did not use the specific characteristics of automotive software systems, we adjusted the metrics without changing their semantics explained by the original authors.

All data used in this study was provided by VCC and was based on several software platforms deployed to different models of Volvo. In order to perform the measurements and present their results, a tool has been implemented which is able to apply the metrics to a large number of software components. After the study, the tool is planned to be used at Volvo before the realization of changes in order to increase the efficiency of the software development process, improve the quality of the system and reduce the production cost.

Theoretical validation of the measures was done according to the complexity and coupling properties defined by Briand et al. [35]. The theoretical validation was based on mathematical models and analysis of these in a theoretical setting. Empirical validation of the metrics was conducted in a car development project at VCC at the time when the project was adopting architectural changes and it was based on the measurement results provided by the implemented tool. Throughout the entire research, many workshops and interviews with system architects, software designers and integration testers were held at VCC. At the beginning, their purpose was to get more familiar with the automotive software development process, system organization and the problems arising from constant changes. Later on, their purpose was to interpret the measurement results and validate them or more exactly check whether the values of the measures reflected the empirical properties of the architecture (e.g., whether a positive change in the complexity measure reflected a positive change in the measured architecture).

The empirical validation was conducted as follows:

1. Extract the data from one version of the architecture and check its validity manually (to avoid measurement errors).
2. Wait for a relevant period of time (decided based on the project schedule) until the number of changes in the architecture accumulates.
3. Extract the data from another version of the architecture and check its validity manually.
4. Compare the extracted data for both versions and list the changes.

5. Interview technical experts in the domain of software architecture and testing in order to check whether the changes listed reflect the changes made in the architecture with respect to complexity and coupling.

The empirical validation was conducted according to the guidelines presented by Fenton and Pfleeger [36].

### 2.4 Designing Software Systems at VCC

Evolutionary changes in automotive software systems (which represent the majority of changes) often involve introduction of new dependency requirements between the existing components. New functions are added based on the existing hardware components and platforms which result in new dependencies between the software components, new code and often increased complexity. The evolution also entails re-factoring of the existing dependencies, e.g., removing dependencies in case they are no longer needed. To explain the measures of size and possible impact of dependency changes to the automotive software systems, we illustrate it with their common hierarchical organization. The hierarchy is important because changes in the higher architectural units and possible faults they might cause usually manifest at different levels which makes it harder to trace them back to the original levels.

The hierarchies in the automotive software development exist in two different views - logical view (where software is designed) and deployment view (where logical software components are allocated to ECUs).

#### 2.4.1 Logical View

The logical view represents a hierarchical organization of software components, sub-systems and domains. Software components are the smallest architectural units grouped into sub-systems mainly according to their functionalities and mutual interaction. In the logical view, the components communicate by sending/receiving logical signals. At the top level, a single automotive software system is usually divided into different domains clustered according to their application area and associated quality requirements [2]. Each domain contains a number of sub-systems which in turn can have different levels of sub-systems and software components. The following domains are the most common ones:

1. **Power train and chassis** contains the sub-systems responsible for controlling the engine, transmission, etc.

2. **Body** contains the sub-systems such as lights, locking, etc.

3. **Safety** contains the sub-systems responsible for active (e.g., cruising, auto-braking) and passive (e.g., air-bags, belts) safety.

4. **Management** contains the common vehicle sub-systems used by all domains, e.g., settings and diagnostics.
5. Human-Machine Interface contains the sub-systems responsible for vehicle users vehicle interaction.

6. Infotainment contains the information and entertainment sub-systems such as navigation, telephone, etc.

Figure 2.1 shows an example of the logical view for one small part of the system containing one domain (SafetyControl), two sub-systems (PedestrianDetection and SafetyHandler) and three software components (PedestrianDetector, PedestrianManager and SafetyBrakeManager). Both the PedestrianDetection and SafetyHandler sub-systems belong to the SafetyControl domain. The PedestrianDetector and PedestrianManager software components belong to the PedestrianDetection sub-system while the SafetyBrakeManager software component belongs to the SafetyHandler sub-system. The example is created solely for the purposes of this paper in order to better explain the common organization of automotive software systems and does not reflect a part of a real system used at VCC.

![Figure 2.1: Example of the logical view](image)

Considering the organization of the logical view explained in this section, a change can be one of the following: (i) addition/removal of a signal between the existing logical software components, (ii) addition/removal of a logical software component with its signals, (iii) addition/removal of a sub-system with its components, (iv) addition/removal of an entire domain\(^3\) with its sub-systems or (v) move of software components/sub-systems between sub-systems/domains.

\(^3\)Note that the addition/removal of domains is not very common during the life-span of one platform.
2.4.2 Deployment View

The deployment view has two purposes: first to show the network topology of ECUs and second to show the deployment of the software components to particular ECU. Different ECUs are connected via electronic system buses (mostly CAN, LIN, MOST and Flex-ray), and they communicate in order to realize one functionality [2]. Domain ECUs are connecting different logical domains and they usually exchange signals via one (backbone) Flex-ray bus. ECUs inside one domain usually communicate via CAN or LIN buses. MOST is used for the infotainment domain due to its high speed capabilities.

Figure 2.2 shows an example of the network topology containing two domain ECUs (SafetyMaster and InfotainmentMaster) and five other ECUs (Radar, Camera, NightVision, TV and Radio). The SafetyMaster and InfotainmentMaster as domain ECUs are connected via Flex-ray electronic bus. The Radar, Camera and NightVision ECUs belong to the SafetyMaster domain. The Radar and Camera are connected via CAN bus while the NightVision is connected to the Camera via LIN bus. The TV and Radio ECUs belong to the InfotainmentMaster domain and they are connected via MOST.

Each software component in the logical view is realized by one software component in the deployment view in one ECU. Many logical software components may be realized by one deployment software component. Often, the decision where one component is deployed is not made according to its functionality, but due to other reasons such as vicinity to hardware (sensors, actuators and buses) or bus load. That is why logical software components from one sub-system may be deployed to different ECUs, and logical software components from different sub-systems may be deployed to the same ECU. Components deployed to different ECUs communicate via ports by sending/receiving system signals.

Figure 2.3 shows an example of the deployment view for the logical view shown in Figure 2.1. Logical software components PedestrianDetector, Pedes-
2.5 Quality Metrics

When an architectural change occurs, it is important to assess the level of complexity and coupling change in the system since it directly affects its maintainability attributes such as flexibility, extensibility and system life time. Indirectly, it affects other quality attributes as well such as reliability (the risk of faults) and robustness (the risk of "ripple" effects). Having in mind the organization of automotive software systems described in Section 2.4, we decided to measure the structural complexity and coupling of modules based on the strength of their dependencies. This process can be completely automated in case of unified approach used by OEMs to store dependency requirements between modules.

We define our complexity and coupling measures according to the properties of the complexity and coupling measures defined in [35] (explained more detailed in Section 2.8.1). Generally, the complexity of one component captures the strength of its dependencies towards other components in the system regardless of the modules they belong to. On the other hand, the coupling of one component captures only the strength of its dependencies towards com-

---

trianManager and SafetyBrakeManager are mapped to the deployment software components with the same names. The PedestrianDetector software component is deployed to the Radar while the PedestrianManager and SafetyBrakeManager software components are deployed to the SafetyMaster ECU.

Figure 2.3: Example of the deployment view

Considering the deployment view organization of the automotive software system explained in this section, a change can be one of the following: (i) addition/removal of a signal between the existing deployment software components, (ii) addition/removal of a deployment software component with its signals, (iii) addition/removal of an ECU with its components and (iv) deployment of software components onto separate ECUs.
ponents belonging to different modules. Despite the fact that the complexity and coupling measures are absolute, our intention is to use them relatively in order to observe the increase/decrease in the modules complexity and coupling over different system releases.

The reason behind this is the fact that different parts of the system may have very different complexities and therefore it is hard to say if a specific complexity and coupling value of one module is satisfactory or not. However comparing the complexity and coupling values between different releases indicate how much has been changed in a particular module. In case a new module is introduced in the system, its absolute complexity and coupling values can be comparing with the complexity and coupling values of other well known modules i.e., utilizing analogies. Additionally, comparisons between the results of two metrics in the same release are also taken into consideration when defining the future strategies for securing the quality requirements of the system (explained more in Section 2.6).

Since automotive software systems can be observed from two different views - logical view and deployment view, both complexity and coupling measures can be applied to both views. In Section 2.5.1 we present the logic behind the logical view complexity and coupling measures. Due to their similarity, in Section 2.5.2 we present the necessary modifications in order to apply them to the deployment view.

2.5.1 Logical View Measures

One of the most suitable structural complexity metrics focused on inter-module complexity is the one defined by Henry et al. based on modules fan-in and fan-out [26]. Fan-in represents the number of modules which are calling a given module while fan-out represents the number of modules which are called by the given module. Complexity of one module is then defined as:

$$C(i) = [\text{fin}(i) \cdot \text{fout}(i)]^2 \quad (2.1)$$

where $\text{fin}(i)$ represents fan-in of module $i$, $\text{fout}(i)$ fan-out of module $i$ and $C(i)$ complexity of module $i$.

Since automotive software systems are distributed, it is not possible to call one module (in our case software component) from another, but rather send and receive signals containing information. However since the main logic based on the number of dependencies stays the same, fan-in can be defined as the number of received signals from the other software components in the system (input complexity) and fan-out as the number of transmitted signals to the other software components in the system (output complexity).

Based on the logic of $\text{fin}$ and $\text{fout}$, we define $\text{cin}$ and $\text{cout}$ to be the input and output complexities of one software component based on complexity attributes (not just the number of sent/received signals) such as hierarchical level of signals, timing constraints, etc. (this is the main adjustment to the original formula). Additionally, we omit the exponent 2 from formula 2.1 due to its unjustified amplification of measurement results. Now we can calculate the complexity of a single component using formula 2.2 and the overall system (sub-system, domain or entire system) complexity with $n$ software components $C(n)$ as a sum of all components’ complexities using formula 2.3:
2.5. QUALITY METRICS

\[ C(i) = \text{cin}(i) \times \text{cout}(i) \]  

(2.2)

\[ C(n) = \sum_{i=1}^{n} C(i) \]  

(2.3)

Please note that according to formula 2.2, a component has a complexity greater than 0 only if it sends at least one signal to another component in the system and receives at least one signal from another component in the system (i.e., has both fan-in and fan-out). The justification for this lies in the distributed implementation of software components in the automotive domain where different components are developed by different suppliers. This implies that two-way dependencies are significantly more dangerous than one-way dependencies since it is harder to coordinate the work and maintain the system, e.g., no sequential development, integration and testing is possible.

Due to the size of automotive software systems, measuring the overall system complexity increase does not provide very useful result since a small change in one part of the system does not significantly affect the entire system. This is why another approach concerning specific inter-component and inter-sub-system dependencies needs to be applied. One of the solutions is to use the Dependency Structure Matrix as defined in [37] in order to present the strength of relations between different components in the system. However for the same purpose, we decided to use graphs in order to simplify the description of the measures. By using partitioning of software components inside the graph, it is also possible to represent the hierarchy of modules (sub-systems and domains) including inter-module and intra-module dependencies.

We define \( G = (S, D) \) to be the graph where \( S \) represents a set of software components and \( D \) a subset of \( S \times S \) representing the directed dependencies between them. An element of \( D \) is then defined as \( d = (s, s', w) \) where \( s \in S \) represents a software component sending a signal (or more signals) to a software component \( s' \in S \), and \( w \) represents the weight assigned to that directed dependency. The weight of the dependency depends on both quantitative (such as the number of signals sent from \( s \) to \( s' \)) and qualitative factors (such as the hierarchical level of signals) and its value is derived from a formula calculating the strength of dependency between the two components (from now on referred to as the “weight formula”).

The weight formula takes into the account several factors: First, the more signals one software component sends to another component, the stronger dependency we have. Then, a new dependency on a higher hierarchical level (e.g., between domains) increases the complexity more than a new dependency on a lower hierarchical level (e.g., between sub-systems inside one domain), and therefore the weight must be higher as well. There are two reasons for this: (i) different domains (e.g., Safety and the Infotainment domains) are usually designed, implemented and tested as independently as possible by different teams and therefore the communication between them is more demanding and (ii) all domains represented by the domain ECUs are usually connected via one backbone bus (as shown in Figure 2.2) which implies the need for higher bus speed and demands more difficult routing of signals between domains (more ECUs need to gateway the signal to its destination). Finally, if other
attributes such as signal timing properties (period, maximum travel time, etc.) are available, they can also be included into the weight formula.

It stems from the previous that the weight formula can contain multiple attributes (the number of exchanged signals, their hierarchical level, timing properties, etc.). Since not all attributes have the same range of values (e.g., the number of exchanged signals between software components is usually 1-10 while the signal period can be between 1-1000 milliseconds), it is necessary to scale them to the desired range with lower limit set to one. This is important because in all scenarios, an attribute should not decrease the weight of the dependency. On the other hand, the number of exchanged signals (which is main and as such mandatory attribute) can have zero value if there is no dependency between two software components. For some attributes without a range (such as the type of signals), it is necessary to include a weight factor in the formula (inter-sub-system signals weight more than intra-sub-system, etc.). In the logical view, we focus on the following two attributes: the number of exchanged signals between software components and their type (intra-sub-system, inter-sub-system or inter-domain). Therefore, the weight formula for calculating weight \( w \) of the directed dependency between software components \( s \) and \( s' \) is defined as follows:

\[
w(s, s') = \sum_{i=1}^{l} \text{type}[\text{sig}_{s \rightarrow s'}(i)]
\]  

where \( l \) represents the number of signals sent from the software component \( s \) to the software component \( s' \) and \( \text{type}[\text{sig}_{s \rightarrow s'}(i)] \) weight of the signal \( i \) depending on its type (intra-sub-system, inter-sub-system, inter-domain). Based on the assessment of the system architects and integration testers from Volvo, it was concluded that the weight of intra-sub-system signals should be 1, inter-sub-systems signals 1.3 and inter-domain signals 1.8 (see Section 2.8.2 for the assessment of different weights). Since an addition of a new signal between two software components cannot reduce the weight of their dependency, signal weight has to be a non-negative number.

After calculating the weights for all dependencies in the graph, we can calculate the input and output complexity for each software component in the system. For example, the sum of all dependency weights from a software component \( s \) to other software components in the system represents output complexity of \( s \), while the sum of all dependency weights from other software components in the system to the software component \( s \) represents input complexity of \( s \).

\[
cout(s) = \sum_{s', s'' \in S \land (s', s'', w) \in D \land s' = s} w(s', s'')
\]

\[
cin(s) = \sum_{s', s'' \in S \land (s', s'', w) \in D \land s'' = s} w(s', s'')
\]  

(2.5)

Applying formula 2.5 to formulas 2.2 and 2.3, the single software component complexity \( C(s) \) and the total complexity of a sub-system, domain or entire system containing \( n \) software components \( C(n) \) can be calculated as follows:
2.5. QUALITY METRICS

\[ C(s) = \sum_{s', S(s') \in S \land (s', s', w) \in D \land s'} w(s', s'') \times \sum_{s', S(s') \in S \land (s', s', w) \in D \land s' = s} w(s', s'') \]  
\[ (2.6) \]

\[ C(n) = \sum_{s \in S} C(s) \]  
\[ (2.7) \]

According to formula 2.7, the explained complexity measure includes the internal module dependencies inside sub-systems and domains when calculating their complexity. However, the measure excluding internal dependencies could also be useful, especially for predicting possible fault propagation in the system. For this purpose, the Package Coupling Metrics (PCM) named and defined by Gupta et al. can be used to supplement the explained complexity measure [27]. According to Gupta et al., formulas 2.8 and 2.9 can be used to calculate the coupling between two packages \( P_a \) and \( P_b \) on a hierarchical level \( l \) (denoted as \( Coup(P_a^l, P_b^l) \)) and the total coupling of a single package \( P_a \) (denoted as \( PCM(P_a^l) \)), respectively. The calculation is done based on the number of dependencies between software components contained inside the packages (where one component belongs to one package, and the other component belongs to the other package on the same hierarchical level).

\[ Coup(P_a^l, P_b^l) = \sum_{i=1}^{n} \sum_{j=1, j \neq i}^{m} r(e_i^{l+1}, e_j^{l+1}) + \sum_{j=1}^{n} \sum_{i=1, i \neq j}^{m} r(e_j^{l+1}, e_i^{l+1}) \]  
\[ (2.8) \]

\[ PCM(P_a^l) = \sum_{b=1, b \neq a}^{t} Coup(P_a^l, P_b^l) \]  
\[ (2.9) \]

where \( P_a^l \) and \( P_b^l \) represent two packages on hierarchical level \( l \), \( r(e_i^{l+1}, e_j^{l+1}) \) directed dependency between module \( e_i \) and module \( e_j \) on hierarchical level \( l + 1 \) (where \( e_i \in P_a^l \) and \( e_j \in P_b^l \)), \( m \) and \( n \) their total number of components, respectively, and \( t \) total number of packages in the system.

Applied to the logical view hierarchy, graph \( G \) can be used as source for obtaining strengths of dependencies between software components belonging to different sub-systems/domains since sub-systems/domains can be represented as different partitions of \( S \) in \( G \). Therefore, we can introduce a new set \( P \) which represents the set of all partitions in \( G \) such that all of its elements are disjoint and the union over \( P \) equals \( S \). Then the following formulas can be used to calculate the package coupling \( Coup(p, p') \) between two sub-systems/domains \( p \) and \( p' \) and the total coupling \( PCM(p) \) of a single sub-system/domain \( p \):

\[ Coup(p, p') = \sum_{s, s' \in S \land (s, s', w) \in D \land s \in p \land s' \in p'} w(s, s') \]

\[ + \sum_{s, s' \in S \land (s', s, w) \in D \land s \in p \land s' \in p'} w(s', s) \]  
\[ (2.10) \]
PCM \( (p) = \sum_{p', p'' \in P \land p = p'} Coup(p', p'') \) (2.11)

The results of the complexity measures defined in formulas 2.6 and 2.7 and the coupling measures defined in formulas 2.10 and 2.11 should be compared and analyzed together, as explained in Section 2.6. An example of the measurement process is presented in Section 2.7.

### 2.5.2 Deployment View Measures

When creating the requirements specification for suppliers for a particular functionality, the information from the logical view is not sufficient. The suppliers need to know to which ECUs a particular software component will be deployed as well. This is due to the existence of several external requirements such as hardware requirements (CPU load, memory consumption, etc.) which needs to be known by the suppliers when implementing software components. For this reason, it is also important to estimate the potential impact of changes to the network topology in the deployment view. This is done by measuring the complexity and coupling increase in the system in a similar way as it is presented for the logical view in Section 2.5.1.

In the deployment view, logical software components are mapped to deployment software components and ECUs can be considered as sub-systems containing the deployment components. Therefore the structure of the deployment system is logically identical to the structure of the logical system so both the complexity and coupling measures defined for the logical view can be applied here as well. However, formula 2.4 used for calculating the strength of dependencies between software components (the weight formula) has to be modified for two reasons: first, the signal types are no longer intra-sub-system, inter-sub-system and inter-domain, but intra-ECU and inter-ECU instead. Second, an additional timing constraint concerning the maximum allowed time for a signal to travel between ECUs (MaxAge) is available for the system signals (inter-ECU signals) and therefore it should also be included in the weight formula to increase its precision. The lower the MaxAge value is, the more complex system we have since it is harder to satisfy all timing requirements, e.g., a signal must be sent faster which implies that its scheduling on the bus will become more dense.

Based on the assessment of the system architects and integration testers from Volvo, it was concluded that the weight of intra-ECU signals should be 1, inter-ECU signals 1.5 and the weight of MaxAge attribute should vary from [1-1.5] depending on its value (see Section 2.8.2 for the assessment of different weights). Assuming that it ranges from [1-1000] milliseconds (ms), the new weight formula looks as follows:

\[
\begin{align*}
    w(s, s') &= \sum_{i=1}^{l} \text{type}[\text{sig}_{s \rightarrow s'}(i)] \times \{1.5 - \text{MaxAge}[\text{sig}_{s \rightarrow s'}(i)]/2000\text{ms}\} \\
\end{align*}
\]

(2.12)

where \( l \) represents the number of signals sent from the software component \( s \) to the software component \( s' \), \( \text{type}[\text{sig}_{s \rightarrow s'}(i)] \) weight of the signal \( i \) depending on its type (intra-ECU, inter-ECU) and \( \text{MaxAge}[\text{sig}_{s \rightarrow s'}(i)] \) its maximum
allowed time to travel between $s$ and $s'$. Since the addition of a new signal between two software components cannot reduce the weight of their dependency, signal weight has to be a non-negative number. MaxAge for intra-ECU signals is set to 1000 ms by default in order not to affect the dependency weight.

Please note that there are other factors that could be used in order to make the measurements more precise (e.g., signal length or transmitting frequency) in case they are available on a system design level (depending on the process work-flow, they can be defined on an ECU design level as well which is usually done by suppliers using the tools such as Simulink). They can be included in the complexity and coupling calculations in a similar way as explained in formula 2.12. However, if used properly, they should only affect the precision of the presented measures - not their logic.

Based on the weight of all dependencies in the system, the rest of the deployment view complexity and coupling measurements can be done in the same way as explained in the logical view (Section 2.5.1) by applying formulas 2.6 and 2.7 for calculating the complexity and formulas 2.10 and 2.11 for calculating the coupling.

### 2.6 Presentation and Interpretation of Results

#### 2.6.1 Presentation of Measurement Results

Despite the fact that the explained complexity and coupling measures produce numerical results, they are most useful if presented in the context of complexity changes over time. This way system architects, designers and testers can use their knowledge about the system to compare the measurement results with their expectations. In order to maintain the quality of the system during its evolution, the explained measures should be applied before the implementation of each architecturally significant change. The complexity and coupling change should be measured for each hierarchical level in both logical and deployment views. The logical view has three hierarchical levels: logical software components, sub-systems and domains. The deployment system view has two hierarchical levels: deployed software components and ECUs.

Presentation and interpretation of measurement results is crucial for understanding the impact of changes and planning corrective actions in case they are needed. This is why it is important to present the results unambiguously in a simple way so that conclusions can be made quickly. According to [38], using visualizing indicators and their dependencies focused on the most important information is the most efficient way to present the results in a company so that they can be understood easily. Therefore, we decided to use histograms for presenting the complexity and coupling change in the system through different system releases. At the beginning of the histogram, modules with the biggest increase in complexity/coupling should be listed as they pose the biggest threat to the quality attributes of the system. This way, system architects and integration testers can see them instantly and use the visualization to present them to other colleagues as well.

Several histograms are used to present the measurement results and most of them are demonstrated in the example presented in Section 2.7:
1. Logical view software components complexity change presents the change in complexity of all logical software components in the system between two system releases (Figure 2.8).

2. Logical view sub-systems complexity change presents the change in complexity of all sub-systems in the system between two system releases (Figure 2.9).

3. Logical view sub-systems coupling change presents the change in coupling of all sub-systems in the system between two system releases (Figure 2.10).

4. Logical view domains complexity change presents the change in complexity of all domains in the system between two system releases.

5. Logical view domains coupling change presents the change in coupling of all domains in the system between two system releases.

6. Deployment view software components complexity change presents the change in complexity of all deployed software components in the system between two system releases (Figure 2.11).

7. Deployment view ECUs complexity change presents the change in complexity of all ECUs in the system between two system releases (Figure 2.12).

8. Deployment view ECUs coupling change presents the change in coupling of all ECUs in the system between two system releases (Figure 2.13).

Note that it is not possible to measure the coupling of logical and deployment software components as they represent the smallest architectural units.

In addition to the presented histograms which show the complexity and coupling change between two system releases, we suggest the use of Trend charts to present the complexity and coupling change of a specific logical software component, sub-system or domain in the logical view, and the complexity and coupling change of a specific deployed software component or ECU in the deployment view, through all available system releases including the future release (after the realization of changes). This way we can see how big deviation in the complexity and/or coupling of one module is made compared to the initial release. This is especially important having in mind that certain limits are usually set when creating the initial architecture of the system.

2.6.2 Interpretation of Measurement Results

First step when interpreting measurement results is to identify sub-systems, as logical units, and ECUs, as physical units, which have suffered significant increase in the complexity and/or coupling. After identifying such sub-systems and ECUs, we can go one level lower in the hierarchy and see which software components (logical in case of sub-systems and deployment in case of ECUs) are mostly responsible for this increase.

Apart from identifying the sub-systems and ECUs with the biggest change, the focus should be on the comparison between the complexity and coupling measurement results. The following example illustrates why this is important.
Imagine that one sub-system has much higher percentage of complexity increase in comparison to the percentage of coupling increase. This indicates that changes introduced new functionalities assigned to this sub-system but they are mostly local and as such do not represent a big threat to other parts of the system (not high risk of fault propagation and "ripple effects"). Still, this sub-system should be tested more after the implementation of changes in order to assure its quality. On the other hand if one sub-system has the percentage of coupling increase similar to the percentage of complexity increase, this could indicate possible serious architectural changes that may affect many different parts of the system. Therefore the reason and origin of these changes should be investigated further in order to foresee places in the system vulnerable to "ripple effects". The same steps should be taken in case of removal of one sub-system in the logical and ECU in the deployment view (complexity and coupling decrease). In addition to this, every substantial increase in the coupling of domains could be a sign of bad architecture since domains represent the highest logical units in the system which should not be tightly coupled.

If the complexity and/or coupling of one or more parts of the system indicated by the measurement results have increased to unsatisfactory level (decided by system architects and the integration testers which are interpreting the results), it could affect several quality attributes of the system and also the development and testing costs. The impact of the result interpretation to maintainability, robustness, reliability and cost is summarized in Table 2.2:

The level of allowed functional growth, which serves as a base for deciding which complexity and coupling increase is not satisfactory, is usually defined for each part of the system (sub-system or ECU) independently while the initial architecture of the system is created. However, a common belief is that any functional growth larger than 30% in comparison to the initial architecture requires some architectural change such as reallocation of functions to different sub-systems/ECUs. In case the complexity and/or coupling increase is unsatisfactory, there are three possible steps that could be taken in order to minimize the risk of deteriorated quality:

1. Immediate structural re-composition in the parts of the system affected by changes before sending the requirements for their realization to the suppliers. The purpose of this is to balance the complexity and/or coupling in the system (e.g., this could be done by introducing new software components which can take some of the functions.

2. Proceeding with implementation of changes having in mind the identified problems for future system releases or for entirely new software platform.

3. Proceeding with implementation of changes and focusing on the parts of the system with high complexity and coupling increase in the integration/regression testing. This can also reduce the testing cost.

Additionally, the information about the most variable parts of the system could be used to point out functionally unstable sub-systems which need special attention during testing and/or potential structural re-composition in the future releases of the system.
Table 2.2: The impact of the results to the quality attributes and cost

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maintainability</td>
<td>Similar increase in the complexity of one sub-system in comparison to its coupling decreases maintainability since it indicates that the changes are not localized involving more people in the integration and testing activities. This is especially the case if an ECU where most of the components from this sub-system are deployed to shows high coupling increase as it indicates that inter-ECU communication and timing requirements may also be affected.</td>
</tr>
<tr>
<td>Robustness</td>
<td>High coupling increase between two sub-systems indicates that a potential fault introduced by the changes in one sub-system may propagate more easily to the other sub-system decreasing the robustness of the entire system. Therefore, additional testing is needed to reduce the coupling or even a change in the architecture if mostly coupled software components realized by these sub-systems are deployed to different ECUs.</td>
</tr>
<tr>
<td>Reliability</td>
<td>High complexity increase of a particular sub-system/ECU indicates that the reliability of functionalities realized by this sub-system/ECU may be in danger so additional testing of these functionalities is needed.</td>
</tr>
<tr>
<td>Cost</td>
<td>High complexity and coupling increase of one particular ECU is a potential threat to the timing requirements on the electronic buses. This indicates that a different architecture may be needed to balance the effects of changes. Knowing this before the actual implementation of changes is significant for reducing the development cost since it reduces the number of late changes. Additionally, knowing exactly which parts of the system need to be tested reduces the integration testing cost.</td>
</tr>
</tbody>
</table>

2.7 Example

In this section of the paper, we demonstrate the complexity and coupling measurements and show how their results should be presented and interpreted in order to fully capture the impact of changes. We first describe an example system in Section 2.7.1, then we demonstrate the measurements and present their results in Section 2.7.2, and finally we discuss the results in Section 2.7.3.

2.7.1 The Example System Description

We present an example of one part of the automotive software system (from now on referred to as the “system”) from both logical and deployment views. Despite the fact that this system reflects the logic and structure of a real software system used in cars, it does not represent a real system (or part of it) and it is created solely to enhance the understanding of the metrics described in this paper. In order to make it simpler, each logical software component
is deployed to different deployment software component with the same name. This is not necessarily the case in reality where several logical software components are usually deployed to one deployment software component. We used the names ”current” and ”future” system release to denote the design of the system before and after the realization of changes, respectively.

Figure 2.4 shows the logical view of the current system release before the realization of changes. The system is divided into two domains: SafetyControl and VehicleControl. Each domain contains two sub-systems with at least one software component. The purpose of this system is to realize the ”auto-brake” functionality when pedestrian is detected in front of the car.

![Logical View of the Current System Release](image)

Figure 2.4: Logical view of the current system release

The SafetyControl domain is responsible for passengers’ safety inside the car and contains two sub-systems: PedestrianDetection and SafetyHandler. The PedestrianDetection sub-system is responsible for detecting pedestrians on the car’s track and issuing a request for braking to the SafetyHandler sub-system in case a driver did not react fast enough. The SafetyHandler sub-system is responsible for transmitting all safety requests to the VehicleControl domain, such as braking, release of the air-bags, etc.

The VehicleControl domain is responsible for controlling the vehicle and contains two sub-systems: BrakeControl and VehicleManagement. The BrakeControl sub-system is responsible for braking and it periodically sends braking status to the PedestrianDetection sub-system so it can issue a brake request in case a driver is not braking when pedestrian is detected. The VehicleManagement sub-system is responsible for receiving all requests sent to the VehicleControl domain (such as braking and transmission) and forwarding them to the responsible sub-system inside the VehicleControl domain.

Figure 2.5 shows the deployment view of the current system release before the realization of changes. All software components from the logical view shown in Figure 2.4 are now deployed to tree ECUs: SafetyMaster, VehicleMaster and ControlMaster. The number assigned to each inter-ECU signal (system signal) represents its maximum allowed time to travel between two ECUs (MaxAge) in milliseconds.

Figure 2.6 shows the logical view of the future system release after the realization of the changes. The purpose of the changes is to (i) introduce new functionality to the system and (ii) change the used architectural pattern
where all vehicle requests are first sent to the \textit{VehicleManagement} sub-system and then transferred to the right sub-system inside the \textit{VehicleControl} domain. The following changes have been made:

1. A new functionality captured in the \textit{SafetyManager} software component has been introduced to the \textit{SafetyHandler} sub-system. This functionality is responsible for warning the passengers and other road users about the safety issues using sound, lights etc. The information about the safety issues is provided to the \textit{SafetyManager} software component by the \textit{SafetyBreakManager} software component.

2. Along with warning the driver and other road users about the breaking, the \textit{SafetyManager} software component shall also trigger the \textit{BrakeManager} software component directly to perform the actual breaking. Therefore the \textit{VehicleManagement} sub-system inside the \textit{VehicleControl} domain, which was used to collect the information from the \textit{SafetyControl} domain and forward it to the responsible sub-system inside the \textit{VehicleControl} domain, became an unnecessary indirection no longer needed.

3. Since the \textit{SafetyBreakManager} software component is responsible to indicate to the \textit{SafetyManager} software component that breaking is needed, it shall also inform the \textit{BreakStatInformant} software component that break request is issued so it can provide the information to the \textit{PedestrianManager} software component whether or not the driver is breaking.

Figure 2.7 shows the deployment view of the future system release after the realization of changes. The number assigned to each inter-ECU signal represents its MaxAge in milliseconds. The following changes have been made:

1. As logical software components from the \textit{VehicleManagement} sub-system have been removed, the corresponding deployed software components from the \textit{VehicleMaster} and \textit{ControlMaster} ECUs are also removed.

2. It is decided that the \textit{SafetyManager} software component shall be deployed to the same ECU as the \textit{BreakManager} software component (\textit{ControlMaster}). Therefore, a request for breaking sent from the \textit{SafetyManager} software component to the \textit{BreakManager} software component is much faster since it represents an intra-ECU signal.
2.7. Example

3. The new signals between the SafetyBreakManager and SafetyManager software components and the SafetyBreakManager and BreakStatInformant software components are introduced as system signals since these software components are all deployed to different ECUs.

2.7.2 Measurements and Results Presentation

In this section, we demonstrate the use and presentation of the results of the logical and deployment view complexity and coupling metrics based on the example presented in Section 2.7.1.

2.7.2.1 Logical View

In order to calculate the complexity change for each logical software component, sub-system and domain in the system, the weight of all dependencies between software components in both current and future system release shall be calculated first using formula 2.4.

For example in the current system release (Figure 2.4), the PedestrianManager software component is sending one inter-sub-system signal BreakNeeded
to the SafetyBreakManager software component. According to formula 2.4, this implies that the weight of the dependency between the PedestrianManager and SafetyBreakManager software components equals $w = 1.3$ (weight 1.3 for one inter-sub-system signal BreakNeeded).

After calculating the rest of the weights, a complexity of each software component in the current system release can be calculated using formula 2.6. For example, the PedestrianManager software component has an input complexity $cin = 2.8$ (weight 1 for the dependency between the PedestrianDetector and PedestrianManager software components plus weight 1.8 for the dependency between the BreakStatInformant and PedestrianManager software components) and output complexity $cout = 1.3$ (weight 1.3 for the inter-sub-system dependency between the PedestrianManager and SafetyBreakManager software components calculated above). Therefore, its total complexity equals $C(PedestrianManager) = 2.8 \times 1.3 = 3.64$.

After applying the same approach to all software components in both current and future system releases, we can present their complexity change in the histogram shown in Figure 2.8 (the software components are ordered by the complexity difference between the two releases).

![Figure 2.8: Logical software components complexity change](image)

Based on the complexity of all software components in the system, the complexity of each sub-system or domain can be calculated using formula 2.3 as the sum of complexities of all of its components. For example in the current system release (Figure 2.4), the PedestrianDetection sub-system contains two software components: PedestrianDetector with complexity 0 (no input signals) and PedestrianManager with complexity 3.64 (as calculated above). So the complexity of the PedestrianDetection sub-system equals $C(PedestrianDetection) = 3.64$ (0 for the PedestrianDetector software component plus 3.64 for the PedestrianManager software component).

After applying the same approach to all sub-systems in both current and future system releases, we can present their complexity change in the histogram shown in Figure 2.9 (the sub-systems are ordered by the complexity difference between the two releases).
Calculating the complexity difference between the current and future system release for the logical domains can be done in the same way as for the sub-systems, i.e., the complexity change of each domain is calculated as the sum of complexities of all of its components.

In the logical view, the coupling measurements can only be applied to sub-systems and domains (since formula 2.10 requires a package of components). For example, the BrakeControl sub-system in the future system release (Figure 2.6) contains two software components: BrakeManager and BrakeStatInformant. According to formula 2.11, its total coupling is equal to the sum of all dependency weights between these two components and other components in the system, not counting the dependency weight between the two of them. Since the weight of dependencies between the BrakeStatInformant and PedestrianManager software components, the SafetyBreakManager and BrakeStatInformant software components and between the SafetyManager and BrakeManager software components equals 1.8, the coupling of the BrakeControl sub-system equals $Coup(BrakeControl) = 1.8 + 1.8 + 1.8 = 5.4$.

After applying the same approach to all sub-systems in both current and future system releases, we can present their coupling change in the histogram shown in Figure 2.10 (the sub-systems are ordered by the coupling difference between the two releases).

Similarly to this, we can present the coupling change between the current and future system release for all domains using histograms.

The charts shown in figures 2.8, 2.9 and 2.10 are used to present the complexity (for software components) and coupling (for software components and sub-systems) change between the current and future system release. However, it could also be valuable to see the complexity and coupling change of a specific software component, sub-system or domain through all available releases. For this purpose, we suggest the use of Trend charts (example is not shown since we only have two releases in the example).
2.7.2.2 Deployment View

Similarly to the logical view complexity and coupling measurements, first step in measuring the deployment view complexity and coupling change between two releases is to calculate the weight of dependencies between all software components in both current and future system release using formula 2.12.

For example in the future system release (Figure 2.7), the SafetyBreakManager software component is sending two inter-ECU signals (SignalDriver and IssueBreakReq) to the SafetyManager software component. According to formula 2.12, this implies that the weight of the dependency between the SafetyBreakManager and SafetyManager software components equals \( w = 4.125 \) (two inter-ECU signals SignalDriver and IssueBreakReq with weight \( 1.5 \times \frac{1.5250}{2000} = 2.065 \)).

After calculating the rest of the weights, the complexity of each software component in the future system release can be calculated using formula 2.6. For example, the SafetyBreakManager software component has input complexity \( cin = 3.0625 \) (weight 1 for dependency between the PedestrianManager and SafetyBreakManager software components plus weight 2.0625 for the dependency between the SafetyManager and SafetyBreakManager software components) and output complexity \( cout = 6 \) (weight 4.125 for the dependency between the SafetyBreakManager and SafetyManager software components as calculated above plus weight 1.875 for the dependency between the SafetyBreakManager and BreakStatInformant software components). Therefore, its total complexity equals \( C(SafetyBreakManager) = 3.0625 \times 6 = 18.753 \).

After applying the same approach to all software components in both current and future system releases, we can present their complexity change in the histogram shown in Figure 2.11 (software components are ordered by the complexity difference between the two releases).

Based on the complexity of all software components in the system, the complexity change for each ECU can be calculated using formula 2.7 as the sum of complexities of all of its components. For example in the current
system release (Figure 2.5), the SafetyMaster ECU contains only two software components which communicate with other ECUs: PedestrianManager with complexity 2.88 and SafetyBreakManager with complexity 18.36 (as calculated above). Therefore the complexity of the SafetyMaster ECU equals \( C(\text{SafetyMaster}) = 21.24 \) (2.88 for the PedestrianDetector software component plus 18.36 for the PedestrianManager software component).

After applying the same approach to all ECUs in both current and future system releases, we can present their complexity change in the histogram shown in Figure 2.12 (ECUs are ordered by the complexity difference between the two releases).

In the deployment view, the coupling measurements can only be applied to ECUs (formula 2.10 requires a package of components). For example, the
VehicleMaster ECU in the current system release (Figure 2.5) contains two software components: BreakStatInformant and VehicleInfoCollector. According to formula 2.11, its total coupling equals to the sum of all dependency weights between these two components and other components in the system, not counting the dependency weight between the two of them. Since the weight of dependencies between the BrakeStatInformant and PedestrianManager software components and the SafetyBreakManager and VehicleInfoCollector software components equals 1.8, and between the BreakManager and BrakeStatInformant software components and the VehicleInfoCollector and VehicleHandler software components equals 2.06, the coupling of the VehicleMaster ECU equals Coup(VehicleMaster) = 1.8 + 1.8 + 2.06 + 2.06 = 7.88.

After applying the same approach to all ECUs in both current and future system releases, we can present their coupling change in the histogram shown in Figure 2.13 (ECUs are ordered by the coupling difference between the two releases).

![Figure 2.13: Deployment ECUs coupling change](image)

The charts shown in figures 2.11, 2.12 and 2.13 are used to present the complexity (for software components) and coupling (for software components and ECUs) change between the current and future system release. However, it could also be valuable to see the complexity and coupling change of a specific software component or ECU through all available releases. For this purpose, we suggest the use of Trend charts (example is not shown since we only have two releases in the example).

2.7.3 Results Interpretation

We start the analysis with the logical view. Figure 2.9 shows higher increase in complexity of the SafetyHandler sub-system in comparison to its coupling increase shown in Figure 2.10. After looking at the logical software components complexity change presented in Figure 2.8, it can be concluded that there are two main reasons for such a high complexity increase of the SafetyHandler sub-system:
1. The addition of the new software component *SafetyManager*.

2. The increase in complexity of the software component *SafetyBrakeManager* which is a consequence of the new functionality added, i.e., the driver and other road users can now receive sound and visual information when the safety system such as auto-breaking is activated.

Logically, these changes are not truly architectural and can be considered as upgrades of the existing functionality. This means that eventual faults created inside this sub-system during the implementation of the changes are not very likely to affect other parts of the system (system robustness), especially concerning the possibility of fault propagation. However, the *SafetyHandler* sub-system should be thoroughly tested after the integration and possibly broken down into smaller sub-systems in future releases to reduce its complexity.

The removal of the *VehicleManagement* sub-system represents the opposite case. This is a high level architectural change since the *VehicleManagement* sub-system was responsible for receiving all vehicle requests and transferring them to the right sub-system inside the *VehicleControl* domain. Now after the *VehicleManagement* sub-system is removed, the decision about which sub-system inside the *VehicleControl* domain is responsible for receiving a particular request is transferred to the sender side. This change requires identification and testing of all parts of the system involved in the change, in case it is approved for realization by the suppliers.

After determining the cause for substantial complexity increase of the sub-system *SafetyHandler* in the logical view and identifying and approving the removal of the *VehicleManagement* sub-system, we now focus on the impact of these changes to the ECUs in the deployment view. Figure 2.12 and Figure 2.13 show the complexity and coupling change of all ECUs in the system. High complexity and coupling increase of the *SafetyMaster* ECU is expected, due to the new *SafetyManager* software component deployed to it. The same stands for the *ControlMaster* ECU, just this time due to the removal of the *VehicleManagement* sub-system which increases the number of requests received by the software components deployed to the *ControlMaster* ECU (signals can now be received from anywhere, not just from the software components inside the *VehicleManagement* sub-system).

As a consequence of this, the complexity and coupling of the *VehicleMaster* ECU is decreased since not so many signals are sent to the *VehicleManagement* sub-system anymore. In order to approve these changes for realization by the suppliers, it is necessary to verify first that the *SafetyMaster* and *ControlMaster* ECUs can handle the new functionalities and signals on the electronic buses from the hardware perspective (CPU, memory, buses etc).

### 2.8 Validation of the Metrics

#### 2.8.1 Theoretical Validation

In Section 2.5, two different metrics were proposed: the complexity metric based on Henry and Kafura’s Structure Complexity [26] and the coupling metric based on Package Coupling Metric [27]. In this section, we provide
their theoretical validation (proof that they satisfy criteria for the complexity and coupling metrics) according to the complexity and coupling properties defined by Briand et al. [35].

The complexity metric holds all five properties of the complexity metric defined in [35]:

1. **Non-negativity**: The complexity of a system is non-negative.
   
The results of both formulas 2.4 and 2.12 for calculating the weight of dependencies are non-negative values. In formulas 2.6 and 2.7 for calculating the complexity, these values are just summed and multiplied resulting in a non-negative value.

2. **Null value**: The complexity of a system is 0 if there are no relations between its modules.
   
   In case no signals are exchanged between modules in the system, the sum in formulas 2.4 and 2.12 will be zero, resulting in a zero complexity value after applying formulas 2.6 and 2.7.

3. **Symmetry**: The complexity of a system does not depend on the representation of its arcs.
   
   Changing the direction of all signals in the system results in the swapped values of $cin(s)$ and $cout(s)$ defined in formula 2.5. This does not affect the multiplication in formula 2.6 for calculating the complexity.

4. **Module monotonicity**: The complexity of a system is not less than the sum of complexities of its unrelated modules.
   
   The complexity of a system is calculated as a sum of all modules complexities (according to formula 2.7), and as such can not be less than the sum of its unrelated modules.

5. **Disjoint module additivity**: The complexity of a system is equal to the sum of complexities of its disjoint modules.
   
   The same explanation as for "Module monotonicity".

The coupling metric holds all five properties of the coupling metric defined by Briand et al. [9]:

1. **Non-negativity**: The coupling of a system is not negative.
   
   The results of both formulas 2.4 and 2.12 for calculating the weights of dependencies are non-negative values. In formulas 2.8 and 2.9 for calculating the coupling, these values are just summed resulting in a non-negative value.

2. **Null value**: The coupling of a system is 0 if there are no relations between its modules.
   
   In case no signals are exchanged between modules in the system, the sum in formulas 2.4 and 2.12 will be zero, resulting in a zero coupling value after applying formulas 2.8 and 2.9.
2.8. VALIDATION OF THE METRICS

3. **Monotonicity**: The coupling of a system does not decrease with addition of new inter-module relations.

   New inter-module relation increases the coupling in the system if the two modules belong to different packages. Otherwise, there will be no change according to formula 2.8, which validates that it can not decrease.

4. **Merging of modules**: The coupling of a system does not increase when merging two or more of its modules.

   When two or more modules in the system are merged, the coupling will decrease if modules are related and belong to different packages. Otherwise, it will stay the same according to formula 2.8, which validates that it can not increase.

5. **Disjoint module additivity**: The coupling of a system after merging two or more unrelated modules does not change.

   The same explanation as for "Merging of modules".

2.8.2 Empirical Validation

During the entire research, regular meetings were held at VCC on a weekly basis in order to discuss the findings where the system architects, designers and testers from Volvo actively participated in the discussions. After discussing the applicability of the metrics presented in Table 2.1, we concluded that the Henry and Kafura’s Structure Complexity and the Package Coupling Metric are the most applicable metrics, if we include the qualitative signal properties (i.e., the strength of dependency represented by the exchanged signals between software components) and exclude the exponent 2 from the Henry and Kafura’s formula. Therefore, we introduced our weight formula 2.4 for the logical view and weight formula 2.12 for the deployment view and applied Henry and Kafura’s formulas 2.6 and 2.7 for calculating the complexity and Package Coupling Metric formulas 2.8 and 2.9 for calculating the coupling.

The validation of weights presented below shows that by using different signal weights (1 for intra-sub-system, 1.3 for inter-sub-system and 1.8 for inter-domain signals in the logical view and 1 for intra-ECU and 1.5 for inter-ECU signals in the deployment view), we achieve more precise results of the measurements. In case we use weight 1 for all signals, the complexity and coupling calculations are equal to the original Henry and Kafura’s Complexity Metric (without exponent 2) and the Package Coupling Metric, respectively.

In order to validate the results of the metrics based on the complexity and coupling change in the system through different releases, a software tool (QTool) has been implemented. The tool is able to extract the structural data for all logical software components and sub-systems in the logical view, and all deployment software components and ECUs in the deployment view for the chosen platform. Logical domains are not considered in the tool, but they can be approached in the same way as sub-systems. The data is stored internally in order to be able to apply the complexity and coupling metrics when generating the results. The QTool is also able to present the measurement results, as explained in Section 2.6.1.
After extracting the data from two different platform variants (PlatformVar1 and PlatformVar2 which is derived from PlatformVar1 but maintained differently for several years) and applying the metrics based on the two different releases during a relevant period of time (the period was dictated by the car development project schedule/progress), we presented the results to the software architects, designers and tester from VCC in a workshop and came to the results and explanations presented below. The subset of these results showing 10 ECUs (the real ECU names are not showed due to confidentiality) which suffered the biggest increase or decrease in the complexity and coupling in both platform variants are shown in the histograms in figures 2.14, 2.15, 2.16 and 2.17 (they are ordered by their absolute complexity/coupling change with relative complexity/coupling change written in percentage above the bars).

![Histogram of ECU complexity change between two releases](image)

**Figure 2.14: PlatformVar1 ECU complexity change between two releases**

**R1** In both platform variants, the same ECU (named ECUA in Figure 2.14, 2.15, 2.16 and 2.17) suffered the biggest increase in complexity, while its coupling increase was significantly smaller.

**Explanation:** There has been a substantial functional increase of ECUA in both platforms, but most of the new software components and their signals are internal to this ECU. This validates the result described in R1 since internal signaling has impact only on complexity, not coupling.

**R2** In both platform variants, the same sub-system suffered the biggest increase in complexity.

**Explanation:** Most of the software components from this sub-system in both platform variants are deployed to ECUA. This validates the result described in R2 since ECUA has also increased in complexity after these components are deployed to it.
2.8. VALIDATION OF THE METRICS

Figure 2.15: PlatformVar2  ECU complexity change between two releases

Figure 2.16: PlatformVar1  ECU coupling change between two releases

R3 In PlatformVar1, an ECU (named ECUB in figures 2.14, 2.15, 2.16 and 2.17) suffered substantial increase in both complexity and coupling, while in PlatformVar2 it has decreased in both complexity and coupling.

Explanation: In both platform variants, there has been an increase in communication between ECUB and other ECUs in the system. However in PlatformVar2, there has also been a major clean-up of unused signals.
and therefore in the end, the complexity and coupling of this ECU has decreased in this platform in comparison to PlatformVar1. This validates the result described in R3.

R4 In both platform variants, there are several new sub-systems introduced. Explanation: The increase in communication between ECUB and other ECUs in the system in both platform variants comes mostly from the new functionality realized in the new sub-systems containing the software components deployed to ECUB. However in PlatformVar2, some software components are also removed from ECUB due to the clean-up of signals which validates the results described in R3 and R4.

R5 In both platform variants, an ECU (named ECUE in figures 2.14, 2.15, 2.16 and 2.17) suffered increase in both complexity and coupling, but the increase is higher in PlatformVar2 in comparison to PlatformVar1. Explanation: There has been a functional growth of ECUE in both platform variants. However, higher increase in the complexity and coupling is expected in PlatformVar2 since harder timing constraints (signal Max-Age) are set there due to the movement of this ECU from low-speed CAN bus to high speed CAN bus. This validates the result described in R5.

R6 Apart from the first couple of ECUs shown in figures 2.14, 2.15, 2.16 and 2.17, the rest of the ECUs have negligible change in the complexity and coupling (no change or change less than 2%). Explanation: Apart from the mentioned ECUs, there was no major work in other parts of the system which validates the result described in R6.

Based on the presented results, it was concluded that the complexity and coupling metrics can be applied early in the development process before the
realization of changes by the suppliers and that they are able to identify the most complex and coupled parts of the system. It was also concluded that the metrics are able to locate the origin of the most severe architectural changes in the system and present the results in an understandable way. Finally, it was concluded that the suggested interpretation of the measurement results can lead to conclusions which can be used to assure the maintainability, robustness and reliability of the system and reduce the development/maintenance cost.

In order to validate different weights used for intra-sub-system signals (1) and intra-sub-system signals (1.3) in the logical view, and intra-ECU signals (1) and inter-ECU signals (1.5) in the deployment domain, we re-did all the calculations using different weights for all types of signals in both logical and the deployment view and compared the difference in percentage of complexity and coupling change.

In the logical view, we tried three different weights:

1. Weight 1 for both intra-sub-system signals and inter-sub-system
2. Weight 1 for intra-sub-system signals and 1.3 for inter-sub-system signals
3. Weight 1 for intra-sub-system signals and 1.5 for inter-sub-system signals

The results of coupling measurements were identical for all sub-systems since intra-sub-system signals are not taken into the account in the coupling calculation and we did not use different weights for intra-domain signals. However the results of complexity measurements showed different results for ECUs where several or more changes have been made. We also noticed that the order of sub-systems ordered by the complexity difference was changed meaning that some less complex sub-systems became more complex and vice versa. Table 2.3 shows some results of the complexity increase and decrease in percentage using different weights for 5 sub-systems in PlatformVar1 (the real names of sub-systems are not showed due to confidentiality reasons):

<table>
<thead>
<tr>
<th>Subsystem</th>
<th>Weights 1 [%]</th>
<th>Weights 2 [%]</th>
<th>Weights 3 [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SubsystemA</td>
<td>1.44</td>
<td>1.45</td>
<td>1.46</td>
</tr>
<tr>
<td>SubsystemB</td>
<td>-1.35</td>
<td>-1.39</td>
<td>-1.41</td>
</tr>
<tr>
<td>SubsystemC</td>
<td>-25.67</td>
<td>-28.69</td>
<td>-32.00</td>
</tr>
<tr>
<td>SubsystemD</td>
<td>6.13</td>
<td>7.20</td>
<td>7.74</td>
</tr>
<tr>
<td>SubsystemE</td>
<td>251.51</td>
<td>199.19</td>
<td>174.62</td>
</tr>
</tbody>
</table>

For example, SubsystemA and SubsystemB were not affected much by different weights because they did not suffer major changes. However SubsystemC and SubsystemD show bigger difference in percentage of complexity increase (in case of SubsystemC) and complexity decrease (in case of SubsystemD) since several new signals have been added to SubsystemC and several existing signals have been removed from SubsystemD. Additionally, SubsystemE (which had around 180% increase in intra-sub-system signals and around 45% decrease in inter-sub-system signals) showed a substantial decrease in percentage of complexity after increasing the weights of the inter-sub-system signals.
By comparing the results for SubsystemC, SubsystemD and SubsystemE, we can see that depending on the type of change, it is possible that the percentage of complexity change both increases and decreases with the increase in weights. This, together with the fact that the order, by complexity difference, of the top 5 sub-systems (and other sub-systems as well) has changed shows that the decision about different weights for the intra-sub-system and inter-sub-system signals does affect the measurement results.

In the deployment view, we tried three different weights:

1. Weight 1 for both signals intra-ECU signals and inter-ECU signals
2. Weight 1 for intra-ECU signals and weight 1.5 for inter-ECU signals
3. Weight 1 for intra-ECU signals and weight 2 for inter-ECU signal

We got similar results as for the sub-systems in the logical view - no change in the coupling of ECUs since intra-ECU signals are not taken into the account in the coupling calculation, and different complexity values for the ECUs affected by several or more changes. In order to validate that the weights chosen for the complexity calculations are the most suitable weights to reflect the complexity of sub-systems and ECUs, we compared (together with practitioners from VCC in the area of software architecture and integration testing) their complexity change using different weights presented above. The comparison was done with respect to previously estimated complexity of the new/removed functionality and testing effort done by the practitioners in a workshop.

We concluded that weight 1 for intra-sub-system signals, 1.3 for inter-sub-system signals and 1.5 for inter-domain signals in the logical view, and weight 1 for intra-ECU signals and 1.5 for inter-ECU signals in the deployment view are the most suitable weights. However we do not claim that different weights cannot be used to produce valuable results as well.

The tool (QTool) which was used for the data collection, measurement and results presentation is planned to be used as part of the verification process at VCC in order to verify the quality strategies related to the complexity of the automotive software system.

### 2.9 Conclusions

In this paper, we emphasized the importance of change management processes in the development of automotive software systems with a particular focus on the architecturally significant changes. In order to improve the quality of the system (maintainability, robustness, reliability) and its development and maintenance cost, we proposed and evaluated two quality metrics which measure the impact of changes to the complexity and coupling of the system under development. The metrics are based on the already existing and theoretically and empirically validated complexity and coupling measures defined in [26] and [27] but adjusted to fit hierarchical, sub-contractor-oriented organization of automotive software systems seen from two different views - logical and deployment. They are used to support early stages of the development in order to reduce the number of costly and time consuming late changes.
Apart from the description of the measures, we focused on the presentation and interpretation of the measurement results as equally important segments in the decision making process. We proposed a graphical representation of the results using histograms based on the change in the complexity and coupling through different system releases and evaluated it at Volvo Car Corporation. Finally, we stressed that despite the entirely automated measurement process and the presentation of the results, human knowledge about the system and experience play a major role in their interpretation. Common automotive system organization, the complexity and coupling measures, the measurement process and their results presentation and interpretation are all demonstrated in Section 2.7 on the example specially designed for the purpose of this paper.

The presented metrics were theoretically validated according to the complexity and coupling properties defined in [35]. Both metrics and the significance of their results were empirically validated on the software systems used at Volvo cars with the help of software architects, designers and testers from Volvo. However, it is possible that the metrics are applicable to a wider range of software systems which rely on the communication between different system modules by exchanging signals over multiplex buses.

The complexity and coupling metrics described in this paper have some limitations as well. First, the weights of different signals are defined by the practitioners from VCC based on the sample of sub-systems, ECUs and software components from two different platform variants. Therefore, it is possible that for a different sample of sub-systems, ECUs and software components, different weights should be defined in order to keep the desired precision of the metrics. Additionally, the knowledge about the system and the performed changes is needed when interpreting the measurement results in order to distinguish between the expected functional growth of certain parts of the system and potentially dangerous architectural changes.

In our future research, we intend to explore the ways to include the behavioral aspects of automotive software system (e.g., which parts of the system work together in order to fulfill one system functionality) into the account when estimating change impact analysis and compare the results with the results from the structural complexity and coupling metrics described in this paper. One possible solution is to use the Function Point Analysis (FPA) defined in [19] as it can also be applied early in the development process based on the system architecture and design requirements. Additionally, we plan to compare the results of our metrics with the actual number of faults found in a particular sub-system/ECU and testing effort needed to maintain the quality.

The Qtool can be downloaded from the following link: \texttt{http://web.student.chalmers.se/~durisic/QTool.zip}
Bibliography


Chapter 3

Paper B

Evolution of Long-Term Industrial Meta-Models - A Case Study of AUTOSAR

D. Durisic, M. Staron, M. Tichy and J. Hansson

Abstract

Meta-models in software engineering are used to define properties of models. Therefore the evolution of the meta-models influences the evolution of the models and the software instantiated from them. The evolution of the meta-models is particularly problematic if the software has to instantiate two versions of the same meta-model - a situation common for long-term software development projects such as car development projects. In this paper, we present a case study of the evolution of the standardized meta-model used in the development of the automotive software systems – the AUTOSAR meta-model – at Volvo Car Corporation. The objective of this study is to assist automotive software designers in planning long term development projects based on multiple AUTOSAR meta-model versions. We achieve this by performing quantitative analysis of the AUTOSAR meta-model evolution in order to visualize the size and complexity change between different meta-model versions and calculate the number of changes which need to be implemented to adopt a newer version. The analysis is done for each major role in the automotive development process affected by the changes.
3.1 Introduction

The evolution of software today is influenced by the evolution of models and also meta-models. Meta-models are used to define the properties of models and as such they influence the software instantiated from these models [1]. We consider a model as an abstract representation of a software system and a meta-model as a model which defines the syntax and semantics of a particular domain-specific modeling environment [2, 3]. One example of such domain-specific meta-model used in industry is the standardized meta-model used in the development of automotive software systems - AUTOSAR (AUTomotive Open System ARchitecture) [4] meta-model. A simplified example of the usage of the AUTOSAR meta-model to allocate software components onto different Electronic Control Units (ECUs)\(^1\) is presented in Figure 3.1.

![Diagram of AUTOSAR Meta-Model and its usage](image)

Figure 3.1: Example of the AUTOSAR Meta-Model and its usage

As industrial models, like AUTOSAR, are usually exchanged between a number of actors in the development process which may use different tools, meta-models are used as basis for the development of these tools in order to assure tooling interoperability. Therefore the compliance of the models to their meta-models must be preserved to enable different tools to work on the same models. For this reason, the evolution of the meta-models is very important to provide means to express new modeling solutions and as such enable innovation in the software based on these solutions.

In large projects which span over longer period of time (e.g., 4-5 years), new modeling solutions are sometimes needed during the development of one project which implies the co-existence of multiple meta-model versions [5]. The reason for this co-existence is the fact that long life-cycles usually imply the existence of legacy software based on the old meta-model versions but also new software based on the new meta-model versions. This can be observed in

\(^1\)Embedded system (hardware and software) responsible for one or more vehicle functions (e.g., engine control, breaking).
car projects where, due to the distributed nature of the automotive software systems, different sub-systems may have their own development cycles so their models may be instantiated from different AUTOSAR meta-model versions. Under these circumstances, understanding the meta-model evolution is crucial part of project planning.

The solution for the identified problems is to monitor the evolution of the meta-models used in the development projects and analyze them before their adoption. One reason is to better understand possible implications of adopting new meta-model versions based on their size and complexity increase. The other reason is to estimate the effort needed to implement the changes (re-work) based on their number. In this paper, we present a case study analysis of the AUTOSAR meta-model evolution at Volvo Car Corporation (VCC). We first identify the most important roles in the automotive software development process and the relevant types of changes to be considered in the evolution. Then we develop a method for extracting the data from different versions of the AUTOSAR meta-model. Finally we perform quantitative analysis of the data by applying a number of software metrics to visualize the size and complexity trends in the evolution and counting the number of changes between different meta-model versions.

The rest of the paper is structured in the following way: Section 3.2 describes the case study context - automotive software development based on the AUTOSAR standard; Section 3.3 describes the related work; Section 3.4 describes the research goal and the research questions and presents the design of the case study; Section 3.5 presents the results of the case study; Section 3.6 discusses and validates the results of the study and provides recommendation to other companies for monitoring the evolution of their meta-models; finally, Section 3.7 summarizes our conclusions and plans for future work.

### 3.2 Case Study Context

Automotive software systems are distributed systems where one premium vehicle today typically contains 70 - 100 ECUs [6]. Together with their distributed nature, the development of the automotive software systems is also distributed as they are developed in a collaborative environment which involves a number of actors. On one side we have car manufacturers (OEMs - Original Equipment Manufacturers) responsible for designing and verifying the functions and architecture of the system. On the other side we have different layers of suppliers (e.g., application software suppliers, tool suppliers, hardware suppliers) responsible for design, implementation and verification of specific components in the system. In addition to the high complexity implied by distributed development, the complexity of the automotive software systems is constantly increasing [7] due to new functionalities in cars [8].

In order to facilitate the distributed development of automotive software systems, AUTOSAR standard was introduced with the goal to separate the responsibilities of different actors in the process. This separation is based on a three layer software architecture which aims to separate the application software from the underlying basic software (signaling, network management, diagnostics, etc.). Based on this architecture, AUTOSAR provides standard-
ized interfaces between architectural components in order to standardize the exchange format for their models. The models are expressed using XML and the XML schema used for the validation by the AUTOSAR based tools is generated from the AUTOSAR meta-model [9]. A simplified sketch of the AUTOSAR software development process in presented in Figure 3.2.

![AUTOSAR Software Development Process Diagram](image)

Figure 3.2: Automotive software development process based on AUTOSAR

The AUTOSAR meta-model hierarchy is based on the Meta-Object Facility (MOF) standard [10] and it contains 5 meta-layers (4 meta-layers plus MOF). The difference between the classical MOF meta-layers (MOF Mx) and 5 AUTOSAR meta-layers (AR Mx) is that AUTOSAR defines classifiers and their instances (objects) on two different layers while according to MOF they are both defined on MOF M1 (dual classification problem, see [11]). This is depicted in Figure 3.3.

![AUTOSAR - MOF Relation Diagram](image)

Figure 3.3: AUTOSAR - MOF relation

AR M3 (AUTOSAR Profile meta-layer) is based on the UML 2.0 and it defines the used UML stereotypes and annotations. AR M2 (AUTOSAR
Templates meta-layer) defines how to design the automotive electrical system (ECUs, software components, etc.). AR M1 (AUTOSAR User Models meta-layer) represents the actual models developed by the system designers. Finally AR M0 (AUTOSAR User Objects meta-layer) represents the realization of the AUTOSAR models in the actual ECU. In this paper, we analyze the evolution of the AR M2 and the standardized part of the AR M1 meta-layers.

The AR M2 meta-layer consists of a hierarchy of classifiers with their attributes and it is divided into different AUTOSAR ‘templates’. Each template is used to define how to model one specific part of the automotive system (e.g., Software Component template defines software components and their interaction, System template defines communication between ECUs, etc.). The AR M1 meta-layer consists of instances of the AR M2. The instances used for modeling ECU application software are developed by the software designers while the instances used for modeling the configuration of ECU basic software are standardized by AUTOSAR (e.g., COM stack responsible for the ECU communication, I/O responsible for the access to sensors and actuators, Services such as Diagnostics, etc.). In the analysis of the AR M1 evolution, we consider only the standardized part (models of the ECU configuration).

AUTOSAR uses a three digit numbering scheme Rx.y.z to identify different releases which all include a new release of the meta-model. The first digit identifies major releases which are not compatible between each other and should be considered independently. The second digit identifies minor releases which include compatible extensions and bug-fixes. The third digit identifies revisions which usually contain bug-fixes only. The first two digits identify one evolution branch. Maximum two branches may be maintained by the AUTOSAR consortium in parallel where one branch represents a Development branch focused on bug-fixes and innovations, and the other branch represents a Maintenance branch focused mostly on bug-fixes.

3.3 Related Work

There is a lot of research today related to visualization of the software evolution, as presented in the systematic mapping study by Novais et al. [12]. For example Lanza et al. [13] use several object-oriented metrics for visualizing the evolution of classes like us, however they focus on the evolution of source code. Some of the papers are also related to the visualization of the model and meta-model evolution such as the one from Madhavi et al. [14] who propose a framework for visualizing model-driven software evolution or the one from Lange et al. [15] who propose a tool to aid users in tasks such as model understanding, identification of quality problems and evolution trends. However, these papers are analyzing the evolution of entire models without considering their specific parts relevant for different roles. There is also a lack of empirical research in this area, especially related to the visualization of large scale meta-model evolution.

Many papers also present different methods for mining software repositories in the context of software evolution, as presented by Kagdi [16]. For example Zimmermann et al. [17] and Ying et al. [18] build prediction models to predict which classes, functions and attributes will be changed based on
the historical analysis of different source code versions. With respect to meta-model evolution, Vermolen et al. [19] present an interesting research about the coupled evolution of meta-models and models. They propose a method for detecting and formalizing the complex meta-model evolution in order to migrate the existing models according to the new meta-models more easily. However we believe the area of meta-model evolution can also be improved with more empirical studies, especially related to the validation of the proposed methods for re-constructing and monitoring the meta-model evolution on industrial meta-models.

3.4 Case Study Design

We conduct a case study analysis of the AUTOSAR meta-model evolution at VCC based on the guidelines presented by Kitchenham et al. [20] and Runeson et al. [21]. Our research objective is defined according to the structure presented by Wohlin et al. [22] as:

- **Goal:** Analyze the AUTOSAR meta-model evolution.
- **Purpose:** Assist software designers in assessing the size and complexity increase between AUTOSAR meta-model releases and the number of changes to be implemented for adopting a new release.
- **View:** Software designers working with models instantiating multiple AUTOSAR meta-model releases.
- **Context:** Automotive embedded software systems based on the AUTOSAR standard.

In order to achieve this objective, we define the following research questions:

- **Q1:** What is the trend in the size change between AUTOSAR meta-model releases?
- **Q2:** What is the trend in the complexity change between AUTOSAR meta-model releases?
- **Q3:** How many changes need to be implemented to adopt a new AUTOSAR meta-model release?
- **Q4:** Which roles are mostly affected by the evolution of the AUTOSAR meta-model?

In order to provide answers to the research questions, we design our case study analysis around the following 5 steps:

1. Identify roles in the development process which are affected by the AUTOSAR meta-model evolution.
2. Map the identified roles to relevant parts of the AUTOSAR meta-model.
3. Identify the relevant types of changes to be considered in the AUTOSAR meta-model evolution.
4. Define which AUTOSAR meta-model releases shall be considered and extract the relevant data from them.

5. Calculate the metrics on each considered release and visualize the results.

**Step A:** In order to identify the most important roles in the development process based on AUTOSAR, we conducted semi-structured interviews with software engineers from four different companies. We interviewed two engineers from each of two OEM and one engineers from each of two supplier companies. They all have at least ten years of experience with developing automotive software systems and at least five years of experience with AUTOSAR.

**Step B:** We mapped the identified roles to the relevant parts of the AUTOSAR meta-model (changes in these parts affect the mapped roles) in a workshop based on the expert opinion of the AUTOSAR team at VCC (4 software engineers).

**Step C:** In the workshop mentioned in step B, we agreed upon the relevant types of changes to be considered in the analysis based on the analysis of a small sample of changes. We define ‘relevant’ changes as changes which require certain implementation and/or integration effort.

**Step D:** In the workshop mentioned in steps B and C, we agreed upon the set of AUTOSAR meta-model releases which shall be considered in the analysis. We used a meta-data model presented in Figure 3.4 for the extraction of the relevant data from the considered releases. The meta-data model is based on the relevant part of the MOF meta-model.

![Figure 3.4: Meta-data model used for the measurements](image)

Meta-models are divided into Packages which contain Elements - classifiers and instances. The classifier Elements contain Attributes, Connectors of different Type (e.g., Associations, Generalizations) and Annotations describing their additional properties (e.g., regular expressions for strings). The instance Elements contain Connectors (except of Type Generalization as they represent concrete instances) and Annotations (e.g., C type, multiplicities). Finally,
the Connectors and Attributes can also contain Annotations. Each one of the mentioned meta-elements of the meta-data model contains additional properties captured in the attributes of the meta-elements such as Name, Note etc. These properties are also considered when comparing meta-elements between different meta-model releases. To identify meta-elements in different releases, we used their UUIDs (Universally Unique IDentifiers of the objects) except for the Annotations where we used their Name.

Based on the presented meta-data model, we developed a tool to extract the relevant data from the AR M2 and the AR M1 meta-layers designed in the Enterprise Architect tool (used by AUTOSAR meta-model developers). Due to the structure of the AUTOSAR meta-model, Elements in the AR M2 represent classifiers and Elements in the AR M1 represent instances.

**Step E**: In order to measure the properties of the AUTOSAR meta-model evolution, we used the metrics presented in Figure 3.5 driven by the Goal-Question-Metric approach [23]. The chosen metrics are based on the structural object-oriented metrics defined by Genero et al. [24] and Yi et al. [25].

![Figure 3.5: Goal-Question-Metric approach](image)

The Number of elements (NoE) and the Number of attributes (NoA) metrics count the total number of Elements / Attributes respectively in each meta-model release. We use these metrics to measure the increase in size of the meta-model during its evolution as elements (classifiers with their attributes and objects) represent the main building blocks of the AUTOSAR meta-model.

The Number of changes (NoC) metric counts the total number modifications, additions or removals of the meta-elements of the meta-data model. This means that in case one Attribute changed both its Name and its Type, this counts as two changes. Additionally when introducing or removing meta-elements containing other meta-elements (e.g., Elements with Attributes and Connectors), the total number of changes is defined as the total number of modifiable meta-elements contained in the introduced / removed meta-element plus one for the introduced / removed meta-elements itself.
3.5. CASE STUDY RESULTS

For example if one Attribute with three Annotations is removed, this counts as four different changes - one for the Attribute and three for the Annotations. This behavior is justified by the fact that introduction of one Element cannot be counted as one change, like for example a change of Connector’s lower bound, as it requires much more effort to be implemented.

The Number of changed elements (NoCE) and the Number of changed attributes (NoCA) metrics are based on the NoC metric but this time, all modifications, additions and removals of one Element and Attribute respectively count as one change only. We use the NoC, NoCE and NoCA metrics to identify the roles which are mostly affected by the changes and to count the number of changes needed to be implemented to adopt a new AUTOSAR meta-model release.

The Complexity (CPX) metric represents a sum of Henry and Kafura’s structural complexities [26] of all Elements in one meta-model release and it is defined as

\[
CPX(n) = \sum_{i=1}^{n} [\text{FanIn}(i) \times \text{FanOut}(i)]^2
\]

where \( n \) represents a number of Elements and \( \text{FanIn}(i) / \text{FanOut}(i) \) a number of sourceConnectors / destinationConnectors (not counting Connectors of Type Generalization) of the Element \( i \) respectively.

Generally metrics based on fan-in and fan-out are widely used for measuring the structural complexity of modules [7]. Fan-in is defined as the number of modules which are calling a given module while fan-out is defined as the number of modules which are called by the given module. As modules in the AUTOSAR meta-model represent Elements connected by Connectors, it is not possible to call one module from another. However since different Elements may be part of different domains and as such modeled by different teams, any interaction between them can be considered as increase in the overall complexity of the AUTOSAR meta-model. Therefore we consider a sourceConnector as fan-out and a destinationConnector as fan-in property of the Element rather than just its Attribute used for the size measurement.

The Average depth of inheritance (ADIT) metric calculates the average number of parent Elements (connected by Generalization Type of Connectors) for all Elements in one AUTOSAR meta-model release and it complements the CPX metric in measuring the complexity increase between releases.

In order to calculate the metrics on the extracted data from each considered release of the AUTOSAR meta-model, we developed a tool to compare the models of different releases which is also able to visualize the results using line charts, histograms and heatmaps. As Elements in the AR M1 meta-layer represent instances with no Attributes nor Connectors of type other than aggregation (containers aggregating parameters), the NoA, NoCA, CPX and ADIT metrics are not applicable to this meta-layer.

3.5 Case Study Results

In this section, we present the results of the case study structured according to the steps in the case study design.
3.5.1 Identified roles

Based on the interviews with software engineers from VCC, we identified the following roles in the automotive software development process (our objective was to capture the most important roles but other roles may exist too):

- **Application software designers** - a team at the OEMs responsible for designing vehicle functions by defining software components and their interaction (e.g., data exchange points between components).

- **ECU communication designers** - a team at the OEMs responsible for designing the communication between ECUs (e.g., creation of signals and their transmitting on the electronic buses).

- **ECU basic software configurators** - a team at the OEMs responsible for specifying different basic software configuration possibilities (e.g., setting configuration parameter values after building the ECU software).

- **Basic software designers** - a team at the basic software suppliers responsible for designing the basic software modules (e.g., interfaces between the modules, supported services, etc.).

- **ECU communication configurators** - a team at the application software suppliers responsible for configuring ECU communication related basic software modules.

- **Diagnostics configurators** - a team at the application software suppliers responsible for configuring diagnostics related basic software modules.

- **Upstream mapping tool developers** - a team at the tool suppliers responsible for deriving parts of the ECU configuration (i.e., parameter values) from the system models ("upstream mapping").

3.5.2 Mapping of roles

In the workshop with the AUTOSAR team at VCC, we mapped the identified roles to the relevant parts of the AUTOSAR meta-model (if they are affected by the changes in these parts). The outcome is presented in Figure 3.6 ('X' denotes that the corresponding role is affected by the changes in the corresponding part of the meta-model). We use this mapping to analyze the results for the identified roles separately.

We identified that the mapping of roles to meta-model parts is not 1:1. This means that several roles may be affected by the changes in the same part of the AUTOSAR meta-model and also that changes in different parts may affect the same role. We also identified that not all parts of the meta-model are covered by the identified roles, in particular the Methodology, the EcuResourceTemplate and non-communication and non-diagnostic parts of the AR M1. As the Methodology part is auxiliary, the EcuResourceTemplate is not currently used and other non-communication and non-diagnostic parts of the AR M1 are relevant only to specific roles which do not have a significant impact on the development process, we decided to exclude them from the analysis even though they may be relevant for some additional roles.
3.5. CASE STUDY RESULTS

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Part of the meta-model</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AR M2</td>
<td>GenericStructure</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AR M2</td>
<td>CommonStructure</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AR M2</td>
<td>SWComponentTemplate</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AR M2</td>
<td>SystemTemplate</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AR M2</td>
<td>ECUCParameterDefTemplate</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AR M2</td>
<td>ECUCDescriptionTemplate</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AR M2</td>
<td>BswModuleTemplate</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AR M1</td>
<td>COM-Stack</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>AR M1</td>
<td>Mode_Mgm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>AR M1</td>
<td>Services (only diagnostic)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Figure 3.6: Mapping of roles to meta-model parts

3.5.3 Relevant types of changes

By examining a small sample of the AUTOSAR meta-model changes between R4.0.3 and R4.1.1 with the AUTOSAR team at VCC, we realized that several types of changes which belong to the relevant parts of the meta-model are not relevant to any of the mapped roles. We considered a change not to be relevant if it does not require any implementation / integration effort, such as editorial changes in the Notes of the Elements / Attributes or change in the format of the Annotations. Therefore we decided to exclude from the analysis the changes to the Notes of the Elements / Attributes / Connectors. We also decided to consider only the changes to the Annotations

- of type 'obsolete' (the Element will be removed in future),
- related to different configuration times of the ECU configuration parameters (e.g., before / after the ECU software has been built) and
- related to the identification of the system model Elements from which the ECU configuration parameter values are derived (upstream mapping).

In order to validate our assumption that it is necessary to exclude the changes which are not relevant from the analysis, we compared the results of several metrics considering all and considering the relevant changes only. Figure 3.7 shows an example of the comparison between the number of all changes and the number of relevant changes in R4 (i.e., releases R4.x.y).

We can see that the number of all changes between R4.0.2 and R4.0.3 was less than the number of all changes between R4.0.1 and R4.0.2 even
though the number of relevant changes only between $R4.0.2$ and $R4.0.3$ was increased in comparison to the number of relevant changes only between $R4.0.1$ and $R4.0.2$. This behavior is explained by many editorial (irrelevant) changes between $R4.0.1$ and $R4.0.2$. As there are other similar cases to this, we concluded that the results considering all and considering relevant changes only differ quite a lot. This validates our assumption that wrong conclusions can be derived from the results considering all changes in the AUTOSAR meta-model.

### 3.5.4 Considered releases

In a workshop with the AUTOSAR team at VCC, we agreed to consider only the AUTOSAR meta-model releases presented in green in Figure 3.8. Apart from these releases, there are three additional release branches in the beginning of AUTOSAR ($R1.0$, $R2.0$ and $R2.1$) which we decided not to consider for two reasons. First, they are not used today. Second, the release process back then was quite different (releases occurred after every change or a small group of changes), plus the maturity of the meta-model was not as good as in $R3.0.1$ and onwards. Additionally we decided not to consider releases $R3.0.4$ - $R3.0.7$ as their maintenance was negligible due to the fact that most of the AUTOSAR partners quickly moved to release branch $R3.1$.
3.5.5 Measurement results

In this section, we present and analyze the results of the measurements applied on the considered set of AUTOSAR meta-model releases with respect to the research questions.

**Q1: Size trend**: In order to measure the trend in the size increase between different releases of the AUTOSAR meta-model, we compare the results of the **NoE** and the **NoA** metrics for all considered releases. Figure 3.9 shows the number of relevant **Elements** per each minor release / revision in R3 (i.e., releases R3.x.y) and R4 (i.e., releases R4.x.y).

![Figure 3.9: Number of elements - R3 and R4](image)

We can see a much higher increase in size between different major releases (R3 and R4) and also between minor releases (e.g., R3.1.5 to R3.2.1 and R4.0.3 to R4.1.1) in comparison to revisions. Similar results can be seen by comparing the number of **Attributes** between different releases which is also true for the analysis of the identified roles separately.

**Q2: Complexity trend**: In order to measure the trend in the complexity increase between different releases of the AUTOSAR meta-model, we compare the results of the **ADIT** and the **CPX** metrics for all considered releases. The results of the **ADIT** metric for different roles in R3 are stable and the same is true for the results of the **CPX** metric (except a small decrease between R3.0.1 and R3.0.2 and a small increase between R3.2.1 and R3.2.2 affecting mostly the **Application software designers** role). However in R4 (see Figure 3.10), we can see a much higher increase in the results of both the **CPX** and the **ADIT** metrics (more than double increase between R4.0.1 and R4.1.1) which is mostly related to the introduction of new concepts in R4. This is also the reason for 3-5 times higher increase in the results of the **CPX** and the **ADIT** metrics between the releases in R3 and R4.

**Q3: Number of changes**: In order to count the changes needed to be implemented to adopt a new AUTOSAR meta-model release in one project, we compare the results of the **NoC**, **NoCE** and the **NoCA** metrics for all considered releases. Figure 3.11 shows the total number of changes between each two releases of the AUTOSAR meta-model.

We can see that more changes are made between the releases in branch R3.2 and the releases in branch R4.0 / R4.1, than between the releases in branch R3.0 / R3.1 and the releases in branch R4.0 / R4.1. This indicates that the
changes done in branch R3.2 are more than just a subset of the changes done in branch R4.0 / R4.1, even though the Maintenance branch R3.2 should be focused only on bug-fixes and back-porting of the most important concepts from the Development branch R4.0 / R4.1. This means that using later releases in one evolution branch requires more changes to be implemented in order to switch to a release in another evolution branch. By calculating the NoC metric for each role separately, we identified that this is particularly expressed for the ECU communication configurators role whereas other roles are less affected. The NoCE and the NoCA metrics (for applicable roles) show similar results.

**Q4: Affected roles:** In order to identify the roles mostly affected by the evolution of the AUTOSAR meta-model, we compare the results of the NoC, NoCE and the NoCA metrics for all considered releases. Figure 3.12 shows the results of the NoC metric between consecutive releases in R3 and R4 for
Table 3.1: Factors affecting different releases

<table>
<thead>
<tr>
<th>Release</th>
<th>Factors</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3.0.1</td>
<td>Bug-fixes, new AR M1 modules (<em>State Mngr.</em>), new AR M2 templates (<em>BswModuleTemplate</em>), FIBEX standard harmonization</td>
</tr>
<tr>
<td>R3.0.2 -</td>
<td>Bug-fixes only</td>
</tr>
<tr>
<td>R3.0.3</td>
<td></td>
</tr>
<tr>
<td>R3.1.1</td>
<td>Bug-fixes, conc. <em>On-Board Diagnostics (AR M1)</em></td>
</tr>
<tr>
<td>R3.1.2 -</td>
<td>Bug-fixes only</td>
</tr>
<tr>
<td>R3.1.5</td>
<td></td>
</tr>
<tr>
<td>R3.2.1</td>
<td>Bug-fixes, new concepts <em>Partial networking</em> and <em>Production and development errors</em>, back-ported concepts <em>End2End protection</em>, extended</td>
</tr>
<tr>
<td></td>
<td><em>Complex Device Driver, Basic Software Mode Manager</em> and <em>FlexRay ISO Transport protocol</em> modules</td>
</tr>
<tr>
<td>R3.2.2</td>
<td>Bug-fixes only</td>
</tr>
<tr>
<td>R4.0.1</td>
<td>Bug-fixes, new concepts <em>Ethernet, Variant handling, Timing model</em>, etc. meta-model cleanup</td>
</tr>
<tr>
<td>R4.0.2</td>
<td>Bug-fixes, new AR M2 templates (<em>StandardizationTemplate, AutosarTopLevelStructure</em>), new AR M1 module <em>MemMap</em></td>
</tr>
<tr>
<td>R4.0.3</td>
<td>Bug-fixes, forward-ported concept <em>Partial networking (R3.2.1)</em>, new AR M1 module <em>FlexRay AR Transport Protocol</em>, new SPEM UML profile for</td>
</tr>
<tr>
<td></td>
<td><em>Methodology</em></td>
</tr>
<tr>
<td>R4.1.1</td>
<td>Bug-fixes, new concepts <em>Partial networking on Ethernet</em>, continued <em>FIBEX harmonization</em> and <em>Timing model</em>, J1939 for heavy duty vehicles, etc., maintainability improvements (revision of the</td>
</tr>
<tr>
<td></td>
<td><em>ECU vs. Local scope of AR M1 parameters</em>)</td>
</tr>
<tr>
<td>R4.1.2</td>
<td>Bug-fixes only</td>
</tr>
</tbody>
</table>

3.6 Discussion and Validation of the results

In order to validate the conclusions we derived from the measurements, we analyzed the release notes of the considered AUTOSAR releases. The brief summary is shown in Table 3.1.

We identified a substantial increase in the size and the complexity of R4 releases in comparison to R3 and we relate this to many more new concepts incorporated in R4 than in R3. The incorporation of fewer concepts into R3.1 / R3.2 is related to the fact that R3.2 is a Maintenance branch. Generally we see that new concepts are the main drivers of the AUTOSAR evolution and as such they are mostly responsible for the increase in the number of changes, size and complexity of the meta-model. They are also responsible for the higher increase in the number of changes between minor releases (e.g., R3.1.1 to 3.2.1 related to the new concepts *On-Board Diagnostics II* and *Partial networking*).
in comparison to revisions which contain bug-fixes only (e.g., R3.1.2 - R3.1.5, R3.2.2, R4.1.2). Apart from the new concepts, we believe the meta-model cleanup activity between R3 and R4 is the reason for higher increase in the number of changes needed to be implemented for a switch from a release in R3 to a release in R4.

By analyzing different concepts, we concluded that they mostly affect the ECU communication related parts of the AUTOSAR meta-model (e.g., harmonization with the FIBEX standard used to specify the communication between ECUs, Ethernet as a communication medium, Partial networking for partially switching off the communication). Several concepts are also related to the ECU diagnostics (e.g., On-board diagnostics, Production and development errors). This validates our conclusion that the ECU communication configurators and the Diagnostics configurators roles are mostly affected by the AUTOSAR meta-model evolution and need most re-work.

Even though we designed our case study for analyzing the evolution of the AUTOSAR meta-model, we believe that most of the steps are applicable to a wider range or industrial meta-models based on MOF. Therefore we recommend to other companies who would like to monitor the evolution of meta-models used in their development projects the following:

1. Role based analysis of the meta-model by mapping different roles to the relevant parts of the meta-model.
2. Consideration of the relevant changes only. This is applicable only if the meta-model contains data which does not affect the tools working with the models.
3. Usage of the proposed data-model and the metrics for the analysis of the meta-model evolution. Note that not all data-model parts are applicable to all meta-models, e.g., Generalization Connectors in case of flat meta-model structure.

3.7 Conclusions

In this paper, we present a case study analysis of the AUTOSAR meta-model evolution. The goal of the study was to assist software designers who work with multiple AUTOSAR meta-model releases in planning the adoption of newer releases in the development projects. We achieved this by visualizing the size and the complexity increase between different meta-model releases and calculating the number of changes needed to be implemented in order to adopt a newer release. As these results are based on the quantitative data analysis, they can be fully automated and as such used as an early indicator of possible impact of adopting new meta-model releases on the existing projects and used modeling tools and also as a preliminary estimate of the effort needed to implement the changes.

In order to understand possible implications of adopting new meta-model releases, we showed the results of the Number of elements, the Number of Attributes, the Complexity and the Average dept of inheritance metrics for each role in each release. For example, a high complexity increase between current and adopting meta-model release for a certain role may have a substantial
impact on the quality of the corresponding models instantiating these releases. We showed that the size and the complexity of the AUTOSAR meta-model is increasing between different minor and major releases while it is relatively stable between different revisions. The ECU communication configurators role followed by the Diagnostics configurators role is mostly affected by the changes.

In order to estimate the effort needed to switch from one meta-model release to another, we calculated the Number of changes and the Number of changed elements / Number of changed attributes metrics between each two release of the meta-model. We assume that each change requires a certain implementation effort and therefore more changes between two releases indicate higher effort in switching from one release to another. We showed that the highest effort is needed when switching from a late AUTOSAR meta-model release in one evolution branch to a late release in another evolution branch.

In our future work, we plan to study the evolution of the UML 2.0 metamodel using the same approach as described in this paper. We also plan to assess the applicability of different metrics for measuring the evolution of domain-specific meta-models.
Bibliography


Chapter 4

Paper C

Quantifying Long-Term Evolution of Industrial Meta-Models - A Case Study
D. Durisic, M. Staron, M. Tichy and J. Hansson

Abstract

Measurement in software engineering is an important activity for successful planning and management of projects under development. However knowing what to measure and how is crucial for the correct interpretation of the measurement results. In this paper, we assess the applicability of a number of software metrics for measuring a set of meta-model properties - size, length, complexity, coupling and cohesion. The goal is to identify which of these properties are mostly affected by the evolution of industrial meta-models and also which metrics should be used for their successful monitoring. In order to assess the applicability of the chosen set of metrics, we calculate them on a set of releases of the standardized meta-model used in the development of automotive software systems – the AUTOSAR meta-model – in a case study at Volvo Car Corporation. To identify the most applicable metrics, we used Principal Component Analysis (PCA). The results of these metrics shall be used by software designers in planning software development projects based on multiple AUTOSAR meta-model versions. We concluded that the evolution of the AUTOSAR meta-model is quite even with respect to all 5 properties and that the metrics based on fan-in complexity and package cohesion quantify the evolution most accurately.
4.1 Introduction

Measuring the properties of software today is an inseparable part of software engineering. As the results of the measurements may have a severe impact on project decisions, choosing the right properties to be measured and the right metrics for their measurement is crucial for the correct interpretation of the measurement results [1]. One particularly important use of software metrics is for monitoring the evolution of software [2]. As meta-models are used to define properties of models and as such they influence the software instantiated from these models [3], monitoring the evolution of the meta-models plays an important role in planning the evolution of the software based on them. The goal of this paper is to identify the most applicable metrics for effective monitoring of the evolution of the industrial meta-models.

Industrial meta-models represent a specific kind of meta-models as they are used to define domain-specific models [4] (e.g., telecommunication, automotive, avionics) which are usually exchanged between a number of actors in the development process. As these actors may use different tools to work with the models, meta-models are used as basis for the development of these tools in order to assure their interoperability. Therefore the compliance of the models to their meta-models must be preserved to enable different tools to work with the same models. For this reason, the evolution of such product oriented meta-models is very important to provide means to express new modeling solutions and as such enable innovation in the software based on these solutions.

One example of such industrial meta-model is the standardized meta-model used in development of automotive software systems - AUTOSAR (AUTomotive Open System ARchitecture) [5] meta-model. A simplified example of the usage of the AUTOSAR meta-model to allocate software components to Electronic Control Units (ECUs)\(^1\) is shown in Figure 4.1.

---

\(1\) Embedded system (hardware and software) responsible for one or more vehicle functions (e.g., engine control, breaking).
In large projects which span over longer period of time (e.g., 4-5 years), monitoring the evolution of meta-models is even more important as multiple versions of one meta-model may need to co-exist in one project [6]. The reason for this is that long life-cycles usually imply the existence of the legacy software based on the old meta-model versions but also the new software based on the new versions. This can be observed in car projects where, due to the distributed nature of the automotive systems, different sub-systems may have their own development cycles so their models may be instantiated from different versions of the AUTOSAR meta-model. Therefore measuring certain properties of meta-models between different versions is important to understand the potential impact of adopting new meta-model versions in terms of compatibility and effort in updating the existing tools and models.

In this paper, we present the applicability assessment of a number of software metrics for monitoring 5 properties of meta-model evolution - size, length, complexity, coupling and cohesion [7]. We assess the metrics in a case study of AUTOSAR meta-model at Volvo Car Corporation. To identify the most applicable metrics, we used Principal Component Analysts (PCA) [8]. The results of these metrics shall be used by software designers for two main purposes: First, to plan the adoption of new AUTOSAR meta-model releases in on-going or future development projects by providing initial estimations about the adoption effort. Second, to predict the impact of adopting new AUTOSAR meta-model releases on the existing models in terms of quality and re-work.

Based on the PCA results, we concluded that the evolution of the AUTOSAR meta-model is quite even with respect to all 5 properties. We also concluded that the metrics based on fan-in complexity and package cohesion quantify the evolution most accurately. This is validated by comparing the results of these metrics with release notes of each AUTOSAR release.

The rest of the paper is organized in the following way: Section 4.2 describes the related work; Section 4.3 describes the context of the case study - the AUTOSAR meta-model; Section 4.4 describes the design of the case study including the research questions and the research method; Section 4.5 formally defines the assessed metrics; Section 4.6 presents the results of the PCA performed on the results of the metrics calculated on a number of releases of the AUTOSAR meta-model; finally, Section 4.7 summarizes our conclusions and plans for future work.

4.2 Related Work

A number of papers today analyze the evolution of software, especially related to visualization of the software evolution [9]. Some of them focus on the evolution of models, like the one from Madhavi et al. [10], or they define or analyze the metrics applicable for measuring their properties such as the ones from Hyoseob et al. [11], Marchesi et al. [12] and McQuillan et al. [13]. However not many papers focus on the analysis of the meta-model evolution. Additionally, there is a lack of empirical research in this area, especially related to the evolution of long-term industrial meta-model.

For the definition of metrics, we decided to use formalized definition based on the set theory. However there are several other applicable approaches to the
formal definition of object-oriented software metrics such as the one proposed by Baroni et al. using OCL [14], the one proposed by Wakil et al. using XQuery expressions for XMI documents [15] or the one proposed by Lamrani et al. using Z language [16].

Finally, we use PCA to assess the correlations between different metrics and to identify the metrics which are able to measure the desired properties most accurately. This was the goal of several other papers such as the ones from Del Almo et al. [17], Dash et al. [18] and Nagappan et al. [19].

4.3 AUTOSAR Meta-Model and its Role

Automotive software systems are distributed systems where one premium vehicle today typically contains around 70 - 100 ECUs [20]. Together with their distributed nature, the development of the automotive software systems is also distributed as they are developed in a collaborative environment which involves a number of actors. On one side we have car manufacturers (OEMs - Original Equipment Manufacturers) responsible for designing and verifying the functions and the architecture of the system. On the other side we have different layers of suppliers (e.g., application software suppliers, tool suppliers, hardware suppliers) responsible for design, implementation and verification of the specific components in the system. In addition to the high complexity implied by the distributed implementation and development, the complexity of the automotive software systems is constantly increasing [21] due to new features in cars [22].

In order to facilitate the distributed development of automotive software systems, AUTOSAR standard was introduced with the goal to separate the responsibilities of different actors in the development process. This separation is based on a three layer software architecture which aims to separate the application software from the underlying basic software (e.g., signaling, diagnostics). Based on this architecture, AUTOSAR provides standardized interfaces between the architectural components in order to standardize the exchange format for their models between different tools. A simplified sketch of the AUTOSAR software development process in Figure 4.2.

![Figure 4.2: Automotive software development process based on AUTOSAR](image-url)
The AUTOSAR models are expressed using XML and the XML schema used for validation of the models is generated from the AUTOSAR meta-model [23] (see ). Therefore the AUTOSAR meta-model is used as a basis for designing different parts of the AUTOSAR architecture.

The AUTOSAR meta-model hierarchy is based on the Meta-Object Facility (MOF) standard [24] and it contains 5 meta-layers (4 meta-layers plus MOF). Each meta-layer instantiates the layer above, as depicted in Figure 4.3.

![AUTOSAR meta-model layers](image)

Figure 4.3: AUTOSAR meta-model layers

The AR M3 (AUTOSAR Profile meta-layer) is based on the UML 2.0 and it defines the used UML stereotypes and tags. The AR M2 (AUTOSAR Templates meta-layer) defines how to design the automotive electrical system (ECUs, software components, etc.). The AR M1 (AUTOSAR User Models meta-layer) represents the actual models developed by the system designers. Finally the AR M0 (AUTOSAR User Objects meta-layer) represents the realization of the AUTOSAR models in the actual ECU. In this paper, we analyze the AR M2 layer which we refer to as the AUTOSAR meta-model.

The AUTOSAR meta-model consists of a hierarchy of classifiers with their attributes and it is divided into a number of top level packages referred to as AUTOSAR 'templates'. Each template is used to define how to model one specific part of the automotive system (e.g., Software Component Template defines software components and their interaction, System Template defines communication between ECUs, etc.). Classes of the AUTOSAR meta-model may be specialized from multiple classes.

### 4.4 Case Study Design

We conduct a case study analysis [25, 26] of the applicability of a number of software metrics for quantifying the evolution of the AUTOSAR meta-model.
at Volvo Car Corporation. The formal definition of our research objective is defined according to the structure of Wohlin et al. [27] as:

- **Goal:** Assess the applicability of a number of metrics for quantifying a set of meta-model properties.

- **Purpose:** Identify the most applicable metrics for monitoring the AUTOSAR meta-model evolution.

- **Field:** Size, length, complexity, coupling and cohesion properties of the meta-model.

- **View:** Software designers working with models instantiating multiple AUTOSAR meta-model versions.

- **Context:** Automotive software systems based on the AUTOSAR standard deployed to Volvo cars.

In order to extract the relevant data for the measurements from different AUTOSAR meta-model releases, we defined a meta-data model (simplified version of MOF) presented in Figure 4.4.

![Figure 4.4: Meta-data-model used for the measurements](image)

The MetaModel is situated at the top of the hierarchy and it contains a number of top level Packages - called templates. Each template is used to define how to model one specific part of the automotive electrical system, e.g., Generic Structure Template defines generic Classes from which all other Classes are specialized, Software Component Template defines software components and their interaction, System Template defines communication between...
ECUs, etc. The templates contain a hierarchy of Packages where each Package contains Classes and/or other Packages. The Classes contain Attributes. Finally, binary relations between the Classes are realized using Connectors which can be either Generalizations or Associations.

In order to monitor the evolution of the AUTOSAR meta-model, we chose to assess a set of structural object-oriented metrics based on the metrics presented by Genero et al. [28] and Yi et al. [29] as they are applicable to class diagrams which represent building blocks of the AUTOSAR meta-model. We selected 10 metrics presented in Table 4.1. The metrics are categorized according to the 5 properties defined by Briand et al. [7] - size, length, complexity, coupling and cohesion, and they all satisfy the criteria of the corresponding property. Our goal was to cover each property considering only simple (implementation wise) and easily understandable metrics. Also, the goal was to cover all of the elements of the used meta-data-model presented in Figure 4.4.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Abbreviation</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of classes</td>
<td>NOC</td>
<td>Size</td>
</tr>
<tr>
<td>Number of attributes</td>
<td>NOA</td>
<td>Size</td>
</tr>
<tr>
<td>Depth of inheritance</td>
<td>DIT</td>
<td>Length</td>
</tr>
<tr>
<td>Fan-in</td>
<td>FI</td>
<td>Complexity</td>
</tr>
<tr>
<td>Fan-out</td>
<td>FO</td>
<td>Complexity</td>
</tr>
<tr>
<td>Fan-IO</td>
<td>FIO</td>
<td>Complexity</td>
</tr>
<tr>
<td>Package coupling</td>
<td>PCP</td>
<td>Coupling</td>
</tr>
<tr>
<td>Coupling between classes</td>
<td>CBC</td>
<td>Coupling</td>
</tr>
<tr>
<td>Package cohesion</td>
<td>PCH</td>
<td>Cohesion</td>
</tr>
<tr>
<td>Cohesion ration</td>
<td>CR</td>
<td>Cohesion</td>
</tr>
</tbody>
</table>

For the size property, we chose the Number of classes and Number of attributes metrics. Classes represent the main meta-elements of the AUTOSAR AR M2 meta-model as they define the objects used in the actual models instantiating the AUTOSAR meta-model, e.g., ECUs, SoftwareComponents, SystemSignals, etc. Attributes provide additional information about the Classes, e.g., SystemSignal’s length. As the AUTOSAR meta-model does not contain methods and Packages are just logical structures of Classes without any meaning, we consider the number of Classes and the number of Attributes as the most suitable indicators of the size increase of the AUTOSAR meta-model.

Note that even though in the modeling world Associations can be considered as Attributes of the source Classes, in case of industrial meta-models they may have slightly different semantics. The reason for this is the fact that Classes represent logical entities whose instances may be modeled by separate teams. Therefore the introduction / removal of one Association may have globally wider impact than the introduction / removal of one Attribute which describes only one logical entity (Class). For this reason, we analyzed them in a context of complexity, coupling and cohesion rather than in the context of size. Figure 4.5 shows an example of different usage of Associations and Attributes in the AUTOSAR meta-model.
One SoftwareComponent can be allocated onto one Ecu. This allocation is captured in another modeling entity SwcToEcuMapping which contains Associations to both SoftwareComponent and Ecu. Therefore these Associations may introduce additional complexity to both SoftwareComponent and Ecu modeling entities as they may be modeled by separate teams. On the other hand diagAddress Attribute describes just one Ecu entity (it indicates the ID of the Ecu used for responding to diagnostic routines) and therefore does not require interaction between different teams.

For the length property, we chose the Depth of inheritance metric. The reason for this is a deep inheritance hierarchy of Classes in the AUTOSAR meta-model where Classes at the top are abstract Classes used for defining the high level properties of Classes below (e.g., shortName, category, uuid, etc). The non-abstract Classes may have a hierarchy as well.

For the complexity property, we chose the Fan-in, Fan-out and Fan-IO metrics. Generally metrics based on fan-in and fan-out are widely accepted for measuring structural complexity of different modules. Fan-in represents the number of modules which are calling a given module while fan-out represents the number of modules which are called by the given module.

For the coupling property, we chose the Package coupling and the Coupling between classes metrics. Both metrics are based on fan-in and fan-out properties of Classes, just Coupling between classes metric considers all Associations connecting the analyzed Class with other Classes while Package coupling metric considers only the Associations connecting the analyzed Class with Classes from other Packages.

Finally for the cohesion property, we chose the Package cohesion and the Cohesion ratio metrics. Both metrics are based on fan-in and fan-out properties of Classes explained above, just considering only the Associations connecting the analyzed Class with Classes inside the same Package.

Please note that Package cohesion metric is applicable only to meta-models which are well logically structured into different packages according to their functionality rather than according to other properties such as types vs. prototypes, etc. Imagine the case where we have all data-type Classes in one...
4.4. CASE STUDY DESIGN

Package referred to by Classes in other Packages. This results in a low cohesion of these Packages even though the functional cohesion may be high. As we believe the AUTOSAR meta-model is strongly based on the logical units starting with the definition of different templates at the top (see example in Figure 4.6), we decided to include this metric in the assessment even though it may not be a good choice for other industrial meta-models.

<table>
<thead>
<tr>
<th>SoftwareComponentTemplate</th>
</tr>
</thead>
<tbody>
<tr>
<td>- SwcInternalBehavior</td>
</tr>
<tr>
<td>- Runnables</td>
</tr>
<tr>
<td>- VariantHandling</td>
</tr>
<tr>
<td>- ServiceMapping</td>
</tr>
<tr>
<td>- ModeDeclarationGroups</td>
</tr>
</tbody>
</table>

Figure 4.6: Example - Software Component Template package structure

In order to assess the applicability of the analyzed software metrics for monitoring the evolution of meta-models, we study their results on the evolution of the AUTOSAR meta-model. We consider a total number of 22 releases of the AUTOSAR meta-model from the very beginning of AUTOSAR which represents a period of 8 years. The main goal is to eliminate the metrics with redundant results and also to find the metrics which can quantify the evolution of the AUTOSAR meta-model in the most accurate way. In order to achieve this, we performed the PCA to first identify the meaningful principal components and then analyze the importance of the results of each metric in these components. We validated the accuracy of the results of the most important metrics together with the AUTOSAR team at Volvo Cars. We achieved this by comparing the results of the metrics to their expectation based on the analysis of the release notes for each considered AUTOSAR release.

We analyzed the releases of the AUTOSAR meta-model from three different perspectives - the entire AUTOSAR meta-model, Software Component Template and System Template. The Software Component Template and System Template are the two biggest top level packages of the AUTOSAR meta-model in size. For example, the number of Classes of the Software Component Template represents on average 31% of the number of Classes of the entire AUTOSAR meta-model and the number of Classes of the System Template represents on average 30% of the number of Classes of the AUTOSAR meta-model. We also included the Common Structure Template top level package (on average 11% of the number of Classes of the AUTOSAR meta-model) in the analysis of both Software Component Template and System Template packages as its classes are commonly shared between these two templates.
4.5 Definition of the Metrics

The following sub-sections formally define the chosen metrics based on the meta-data-model presented in Figure 4.4.

4.5.1 Number of classes

In order to define the *Number of classes* metric, we first define the following sets:

- \( P(m) = \{p_1(m), p_2(m), ..., p_\alpha(m)\} \) - a set of Packages aggregated by MetaModel \( m \).
- \( P(p) = \{p_1(p), p_2(p), ..., p_\beta(p)\} \) - a set of Packages aggregated by Package \( p \).
- \( C(p) = \{c_1(p), c_2(p), ..., c_\gamma(p)\} \) - a set of Classes aggregated by Package \( p \).

The *Number of classes* metric for Package \( p \) is calculated as a sum of (i) the number of classes aggregated by \( p \) and (ii) the *Number of classes* of the Packages aggregated by \( p \), recursively.

\[
NOC(p) = |C(p)| + \sum_{i=1}^{\left|P(p)\right|} NOC(p_i(p))
\]

The *Number of classes* metric for MetaModel \( m \) is calculated as the *Number of classes* of the Packages aggregated by \( m \).

\[
NOC(m) = \sum_{i=1}^{\left|P(m)\right|} NOC(p_i(m))
\]

4.5.2 Number of attributes

In order to define the *Number of attributes* metric, we first define the following additional set:

- \( A(c) = \{a_1(c), a_2(c), ..., a_\delta(c)\} \) - a set of Attributes aggregated by Class \( c \).

The *Number of attributes* metric for Class \( c \) is calculated as the total number of Attributes aggregated by \( c \).

\[
NOA(c) = |A(c)|
\]

The *Number of attributes* metric for Package \( p \) is calculated as the sum of (i) the *Number of attributes* of the Classes aggregated by \( p \) and (ii) the *Number of attributes* of the Packages aggregated by \( p \), recursively.

\[
NOA(p) = \sum_{i=1}^{\left|C(p)\right|} NOA(c_i(p)) + \sum_{i=1}^{\left|P(p)\right|} NOA(p_i(p))
\]

The *Number of attributes* metric for MetaModel \( m \) is calculated as the *Number of attributes* of the Packages aggregated by \( m \).

\[
NOA(m) = \sum_{i=1}^{\left|P(m)\right|} NOA(p_i(m))
\]
4.5.3 Depth of inheritance

In order to define the Depth of inheritance metric, we first define the following additional set:

- \( C(c) = \{c_1(c), c_2(c), ..., c_\theta(c)\} \) - a set of (‘parent’) Classes connected to Class \( c \) via Generalization Connectors, i.e., target of the Generalization refers to a Class in this set and the source refers to \( c \).

The Depth of inheritance metric for Class \( c \) is calculated as the maximum number of Generalization Connectors in the inheritance hierarchy starting from the considered Class to the (‘root’) Classes with no further parents.

\[
DIT(c) = \begin{cases} 
0, & C(c) = \emptyset \\
\max(\forall c \in C(c) : 1 + DIT(c)), & \text{otherwise}
\end{cases}
\]

The Depth of inheritance metric for Package \( p \) is calculated as the sum of (i) the Depth of inheritance of the Classes aggregated by \( p \) and (ii) the Depth of inheritance of the Packages aggregated by \( p \), recursively.

\[
DIT(p) = |C(p)| \sum_{i=1}^{C(p)} DIT(c_i(p)) + |P(p)| \sum_{i=1}^{P(p)} DIT(p_i(p))
\]

The Depth of inheritance metric for MetaModel \( m \) is calculated as the Depth of inheritance of the Packages aggregated by \( m \).

\[
DIT(m) = \sum_{i=1}^{P(m)} DIT(p_i(m))
\]

4.5.4 FanIn

In order to define the Fan-in metric, we first define the following additional set:

- \( SI(c) = \{si_1(c), si_2(c), ..., si_\varepsilon(c)\} \) - a set of Associations whose target refers to Class \( c \). \( SI \) is short from ‘aSsociation IInput’.

The Fan-in metric for Class \( c \) is calculated as the total number of Associations whose target refers to \( c \).

\[
FI(c) = |SI(c)|
\]

The Fan-in metric for Package \( p \) is calculated as the sum of (i) the Fan-in of the Classes aggregated by \( p \) and (ii) the Fan-in of the Packages aggregated by \( p \), recursively.

\[
FI(p) = |C(p)| \sum_{i=1}^{C(p)} FI(c_i(p)) + |P(p)| \sum_{i=1}^{P(p)} FI(p_i(p))
\]

The Fan-in metric for MetaModel \( m \) is calculated as the Fan-in of the Packages aggregated by \( m \).

\[
FI(m) = \sum_{i=1}^{P(m)} FI(p_i(m))
\]
4.5.5 FanOut

In order to define the Fan-out metric, we first define the following additional set:

- \( SO(c) = \{so_1(c), so_2(c), ..., so_\zeta(c)\} \) - a set of Associations whose source refers to Class \( c \). \( SO \) is short from 'Association Output'.

The FanOut metric for Class \( c \) is calculated as the total number of Associations whose source refers to \( c \).

\[
FO(c) = |SO(c)|
\]

The Fan-out metric for Package \( p \) is calculated as the sum of (i) the Fan-out of the Classes aggregated by \( p \) and (ii) the Fan-out of the Packages aggregated by \( p \), recursively.

\[
FO(p) = \sum_{i=1}^{|C(p)|} FO(c_i(p)) + \sum_{i=1}^{|P(p)|} FO(p_i(p))
\]

The Fan-out metric for MetaModel \( m \) is calculated as the Fan-out of the Packages aggregated by \( m \).

\[
FO(m) = \sum_{i=1}^{|P(m)|} FO(p_i(m))
\]

4.5.6 FanInOut

The Fan-IO metric for one Class is calculated as the multiplication of its FanIn and FanOut values. We chose to multiply Fan-in and Fan-out inspired by the Henry and Kafura’s [30] complexity metric which equals to the squared multiplication of Fan-in and Fan-out. However we decided to remove the square from the formula due to its unjustified amplification of the results (we explained this more in [21]) and because it does not satisfy the criteria of complexity metrics defined in [7] which we used as basis for defining the metrics. The Fan-IO metric for Class \( c \) is defined as:

\[
FIO(c) = FI(c) \times FO(c)
\]

The Fan-IO metric for Package \( p \) is calculated as the sum of (i) the Fan-IO of the Classes aggregated by \( c \) and (ii) the Fan-IO of the Packages aggregated by \( p \), recursively.

\[
FIO(p) = \sum_{i=1}^{|C(p)|} FIO(c_i(p)) + \sum_{i=1}^{|P(p)|} FIO(p_i(p))
\]

The Fan-IO metric for MetaModel \( m \) is calculated as the Fan-IO of the Packages aggregated by \( m \).

\[
FIO(m) = \sum_{i=1}^{|P(m)|} FIO(p_i(m))
\]
4.5.7 Package coupling

In order to define the Package coupling metric, we first define the following subsets:

- \( \text{SIP}(c_x) \subseteq \text{SI}(c_x) \mid \forall s \in \text{SIP}(c_x) : s \in \text{SI}(c_x) \land s \in \text{SO}(c_y) \land c_x \in C(p_x) \land c_y \in C(p_y) \land p_x \neq p_y \) - a subset of Associations whose target refers to Class \( c_x \) aggregated by Package \( p_x \) such that their source refers to Class \( c_y \) aggregated by another Package \( p_y \). SIP is short from 'association Input package coupling'.

- \( \text{SOP}(c_x) \subseteq \text{SO}(c_x) \mid \forall s \in \text{SOP}(c_x) : s \in \text{SO}(c_x) \land s \in \text{SI}(c_y) \land c_x \in C(p_x) \land c_y \in C(p_y) \land p_x \neq p_y \) - a subset of Associations whose source refers to Class \( c_x \) aggregated by Package \( p_x \) such that their target refers to Class \( c_y \) aggregated by another Package \( p_y \). SOP is short from 'association Output package coupling'.

The Package coupling metric for Package \( p \) is calculated as the sum of (i) the total number of Associations whose source / target refers to a Class aggregated by \( p \) and target / source refers to a Class aggregated by another Package, respectively, and (ii) the Package coupling of the Packages aggregated by \( p \), recursively.

\[
\text{PCP}(p) = \sum_{i=1}^{\left| C(p) \right|} (|\text{SIP}(c_i(p))| + |\text{SOP}(c_i(p))|) + \sum_{i=1}^{\left| P(p) \right|} \text{PCP}(p_i(p))
\]

The Package coupling metric for MetaModel \( m \) is calculated as the Package coupling of the Packages aggregated by \( m \).

\[
\text{PCP}(m) = \sum_{i=1}^{\left| P(m) \right|} \text{PCP}(p_i(m))
\]

4.5.8 Coupling between classes

In order to define the Coupling between classes metric, we first define the following additional set:

- \( \text{CP}(c) = \{ cp_1(c), cp_2(c), ..., cp_n(c) \} \) - a set of Classes where there exists an Association whose source / target refers to this Class and target / source refers to \( c \) respectively. CP is short from 'Classes coupled'.

The Coupling between classes metric for Class \( c \) is calculated as the total number of Classes connected to this class via Associations (the source of Association refers to this Class and the target refers to \( c \) or vice versa).

\[
\text{CBC}(c) = |\text{CP}(c)|
\]

The Coupling between classes metric for Package \( p \) is calculated as the sum of (i) the Coupling between classes of the Classes aggregated by \( p \) and (ii) the Coupling between classes of the Packages aggregated by \( p \), recursively.

\[
\text{CBC}(p) = \sum_{i=1}^{\left| C(p) \right|} \text{CBC}(c_i(p)) + \sum_{i=1}^{\left| P(p) \right|} \text{CBC}(p_i(p))
\]
The Coupling between classes metric for MetaModel $m$ is calculated as the Coupling between classes of the Packages aggregated by $m$.

$$CBC(m) = \sum_{i=1}^{\mid P(m)\mid} CBC(p_i(m))$$

### 4.5.9 Package cohesion

In order to define the Package cohesion metric, we first define the following subsets:

- $SIH(c_x) \subset SI(c_x) \mid \forall s \in SIH(c_x) : s \in SI(c_x) \land s \in SO(c_y) \land c_x \in C(p_x) \land c_y \in C(p_x)$ - a subset of Associations whose target refers to Class $c_x$ such that their source refers to Class $c_y$ which are both aggregated by the same Package $p_x$. $SIH$ is short from 'Association Input package cohesion'.

- $SOH(c_x) \subset SO(c_x) \mid \forall s \in SOH(c_x) : s \in SO(c_x) \land s \in SI(c_y) \land c_x \in C(p_x) \land c_y \in C(p_x)$ - a subset of Associations whose source refers to Class $c_x$ such that their target refers to Class $c_y$ which are both aggregated by the same Package $p_x$. $SOH$ is short from 'Association Output package cohesion'.

The Package cohesion metric for Package $p$ is calculated as the sum of (i) the number of Associations whose both source and target refer to a Class aggregated by $p$ and (ii) the Package cohesion of the Packages aggregated by $p$, recursively.

$$PCH(p) = \sum_{i=1}^{\mid C(p)\mid} \mid SIH(c_i(p))\mid + \mid SOH(c_i(p))\mid + \sum_{i=1}^{\mid P(p)\mid} PCH(p_i(p))$$

The Package cohesion metric for MetaModel $m$ is calculated as the Package cohesion of the Packages aggregated by $m$.

$$PCH(m) = \sum_{i=1}^{\mid P(m)\mid} PCH(p_i(m))$$

### 4.5.10 Cohesion ratio

In order to define the Cohesion ratio metric, we first define the following additional subset:

- $CH(c) \subset CP(c) \mid \forall c \in CH(c) : c \in C(p) \land c \in C(p)$ - a subset of Classes coupled to Class $c$ such that they are aggregated by the same Package $p$ which aggregates $c$. $CH$ is short from 'Classes coHered'.

The Cohesion ratio metric for Class $c$ is calculated as a division of (i) the number of Classes connected to $c$ via Associations (the source of the Association refers to this Class and the target refers to $c$ or vice versa) such that they are aggregated by the same Package $p$ which aggregates $c$ and (ii) the number of Classes in $p$. 

$$CR(c) = \frac{\mid CH(c)\mid}{\mid CP(c)\mid}$$
\[ CR(c) = \frac{|CH(c)|}{|C(p)|} : c \in C(p) \]

The Cohesion ratio metric for Package \( p \) is calculated as the sum of (i) the Cohesion ratio of the Classes aggregated by \( p \) and (ii) the Cohesion ratio of the Packages aggregated by \( p \), recursively.

\[ CR(p) = \sum_{i=1}^{\|C(p)\|} CR(c_i)(p) + \sum_{i=1}^{\|P(p)\|} CR(p_i(p)) \]

The Cohesion ratio metric for MetaModel \( m \) is calculated as the Cohesion ratio of the Packages aggregated by \( m \).

\[ CR(m) = \sum_{i=1}^{\|P(m)\|} CR(p_i(m)) \]

### 4.6 Case Study Results

The following section contains the results of the PCA. As input to the PCA, we used the results of the chosen set of 10 metrics calculated on a set of 22 releases of the AUTOSAR meta-model. We performed 3 different PCA based on the results of the metrics calculated on the releases of the (i) entire AUTOSAR meta-model, (ii) Software Component Template package and (iii) System Template package. The results of these 3 PCA are presented in the following sub-sections.

#### 4.6.1 Entire AUTOSAR meta-model

This section presents the results of the Principal Component Analysis (PCA) for which we used as input the results of the chosen set of 10 metrics calculated on a set of 22 releases of the entire AUTOSAR meta-model. Figure 4.7 shows the identified principal components together with the values of their standard deviation, proportion of variance and cumulative proportion of variance. We consider the principal components with the largest proportion of variance as the components which contribute mostly to the results of the metrics, i.e., their significance is the highest.

<table>
<thead>
<tr>
<th>Principal Components</th>
<th>Standard deviation</th>
<th>Proportion of Variance</th>
<th>Cumulative Proportion</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC1</td>
<td>3.0557</td>
<td>0.9337</td>
<td>0.9337</td>
</tr>
<tr>
<td>PC2</td>
<td>0.6711</td>
<td>0.0450</td>
<td>0.9788</td>
</tr>
<tr>
<td>PC3</td>
<td>0.3641</td>
<td>0.0133</td>
<td>0.9920</td>
</tr>
<tr>
<td>PC4</td>
<td>0.2133</td>
<td>0.0046</td>
<td>0.9966</td>
</tr>
<tr>
<td>PC5</td>
<td>0.1507</td>
<td>0.0023</td>
<td>0.9989</td>
</tr>
<tr>
<td>PC6</td>
<td>0.0754</td>
<td>0.0006</td>
<td>0.9994</td>
</tr>
<tr>
<td>PC7</td>
<td>0.0675</td>
<td>0.0005</td>
<td>0.9999</td>
</tr>
<tr>
<td>PC8</td>
<td>0.0356</td>
<td>0.0001</td>
<td>1.0000</td>
</tr>
<tr>
<td>PC9</td>
<td>0.0053</td>
<td>0.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>PC10</td>
<td>0.0001</td>
<td>0.0000</td>
<td>1.0000</td>
</tr>
</tbody>
</table>

Figure 4.7: Principal components - entire AUTOSAR meta-model
The proportion of variances of the identified principal components indicates that the first principal component (PC1) contributes with 93.37% to the variation of the results of the calculated metrics while all the other principal components have significantly less influence. Therefore we concluded that only PC1 is meaningful so we continued with the analysis of the importance of the results of each metric in this principal component. As correlation is generally a good sign of redundancy, we started by investigating the correlation between the results of each two pairs of metrics. Figure 4.8 shows both the importance of the results of each metric in PC1 (table to the left) and the correlation between each two pairs of metrics (table to the right).

By analyzing these results, we concluded that the evolution of the AUTOSAR meta-model is quite even with respect to all five considered properties. We came to this conclusion based on the high correlation between the Number of classes (size), Depth of inheritance (length), Fan-in / Fan-out / FanIO (complexity), Package coupling / Coupling between objects (coupling) and the Package cohesion / Cohesion ratio (cohesion) metrics.

We also concluded that for quantifying the evolution of the AUTOSAR meta-model, it is enough to use only one metric, preferably either the Package cohesion or Fan-in. We came to this conclusion based on the high correlation between the results of all metrics except for the results of the Number of attributes metric which has lower significance. The choice of the Package cohesion or Fan-in metric is based on the highest significance of their results.

### 4.6.2 Software Component Template

This section presents the results of the PCA for which we used as input the results of the chosen set of 10 metrics calculated on a set of 22 releases of the Software Component Template package (including the Common Structure Template). The proportion of variances of the identified principal components is very similar to the results of the PCA for the entire AUTOSAR meta-model. This means that we again identified only one meaningful principal component (PC1) which this time contributes with 96.84% to the variation of the results of the calculated metrics. Figure 4.9 shows both the importance of the results of each metric in PC1 (table to the left) and the correlation between each two pairs of metrics (table to the right).

By analyzing these results, we came to the same conclusions as when analyzing the PCA results for the entire AUTOSAR meta-model - relatively even
4.6. CASE STUDY RESULTS

### Figure 4.9: Metrics correlation - Software Component Template

<table>
<thead>
<tr>
<th></th>
<th>PC1</th>
<th>NOA</th>
<th>FOUT</th>
<th>NOC</th>
<th>CBO</th>
<th>PCP</th>
<th>FIN</th>
<th>PCH</th>
<th>FIO</th>
<th>DIT</th>
<th>CR</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOA</td>
<td>0.1988</td>
<td>NOA</td>
<td>0.0000</td>
<td>0.7269</td>
<td>NOA</td>
<td>0.7181</td>
<td>0.6440</td>
<td>0.3869</td>
<td>0.3825</td>
<td>0.3825</td>
<td>0.3605</td>
</tr>
<tr>
<td>FOUT</td>
<td>0.3159</td>
<td>FOUT</td>
<td>0.7269</td>
<td>1.0000</td>
<td>FOUT</td>
<td>0.9321</td>
<td>0.9801</td>
<td>0.8328</td>
<td>0.8174</td>
<td>0.8174</td>
<td>0.7401</td>
</tr>
<tr>
<td>NOC</td>
<td>0.3282</td>
<td>NOC</td>
<td>0.7181</td>
<td>0.9321</td>
<td>NOC</td>
<td>1.0000</td>
<td>0.9575</td>
<td>0.8775</td>
<td>0.8783</td>
<td>0.8783</td>
<td>0.7686</td>
</tr>
<tr>
<td>CBO</td>
<td>0.3375</td>
<td>CBO</td>
<td>0.6440</td>
<td>0.9801</td>
<td>CBO</td>
<td>1.0000</td>
<td>0.9237</td>
<td>0.9135</td>
<td>0.9135</td>
<td>0.8309</td>
<td>0.7321</td>
</tr>
<tr>
<td>PCP</td>
<td>0.3413</td>
<td>PCP</td>
<td>0.8382</td>
<td>0.8775</td>
<td>PCP</td>
<td>0.9237</td>
<td>1.0000</td>
<td>0.9575</td>
<td>0.9575</td>
<td>0.9575</td>
<td>0.9575</td>
</tr>
<tr>
<td>FIN</td>
<td>0.3429</td>
<td>FIN</td>
<td>0.3825</td>
<td>0.8174</td>
<td>FIN</td>
<td>0.9135</td>
<td>0.9575</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.9426</td>
</tr>
<tr>
<td>PCH</td>
<td>0.3429</td>
<td>PCH</td>
<td>0.8382</td>
<td>0.8775</td>
<td>PCH</td>
<td>0.9135</td>
<td>0.9575</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.9426</td>
</tr>
<tr>
<td>FIO</td>
<td>0.3315</td>
<td>FIO</td>
<td>0.3605</td>
<td>0.7401</td>
<td>FIO</td>
<td>0.9135</td>
<td>0.9575</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.9426</td>
</tr>
<tr>
<td>DIT</td>
<td>0.3050</td>
<td>DIT</td>
<td>0.6555</td>
<td>0.6692</td>
<td>DIT</td>
<td>0.8775</td>
<td>0.8783</td>
<td>0.9237</td>
<td>0.9237</td>
<td>0.9237</td>
<td>0.8862</td>
</tr>
<tr>
<td>CR</td>
<td>0.2906</td>
<td>CR</td>
<td>0.3154</td>
<td>0.5578</td>
<td>CR</td>
<td>0.6846</td>
<td>0.6677</td>
<td>0.8783</td>
<td>0.8783</td>
<td>0.8783</td>
<td>0.8637</td>
</tr>
</tbody>
</table>

4.6.3 System template

This section presents the results of the PCA for which we used as input the results of the chosen set of 10 metrics calculated on a set of 22 releases of the System Template package (including the Common Structure Template). The proportion of variances of the identified principal components is very similar to the results of the PCA for the entire AUTOSAR meta-model and the Software Component Template. This means that we again identified only one meaningful principal component (PC1) which this time contributes with 97.00% to the variation of the results of the calculated metrics. Figure 4.10 shows both the importance of the results of each metric in PC1 (table to the left) and the correlation between each two pairs of metrics (table to the right).

### Figure 4.10: Metrics correlation - System Template

By analyzing these results, we came to the same conclusions as when analyzing the PCA results for the entire AUTOSAR meta-model and the Software Component Template - relatively even evolution of the System Template with respect to all five considered properties where only one metric (Package cohesion or Fan-in) is enough for its successful quantification. In addition to this, we identified that the Cohesion ratio metric, together with the Number of attributes, is not well correlated with the results of other metrics and has lower significance in PC1.
4.6.4 Summary and validation of the metrics results

By analyzing the results of the PCA for the metrics calculated on the entire AUTOSAR meta-model and its two biggest packages, Software Component Template and the System Template, we observed that they are very similar. This is expected as they are based on the same design principles (e.g., logical structuring of Classes into Packages, low coupling between Packages, etc.). Therefore we concluded the following:

1. The evolution of the AUTOSAR meta-model is even with respect to all 5 analyzed properties (size, length, complexity, coupling and cohesion).

2. The correlation between the results of the Number of classes, Depth of inheritance, Fan-in, Fan-out, FanIO, Package coupling, Coupling between objects and Package cohesion metrics is high while the results of the Number of attributes and Cohesion ratio (in case of the Software Component Template and System Template packages) metrics are not very correlated to the results of the other metrics.

3. The results of the Fan-in and Package cohesion metrics are the most accurate for monitoring the evolution of the AUTOSAR meta-model while the results of the Number of attributes and Cohesion ratio (in case of the Software Component Template and the System Template packages) metrics are the least accurate.

These conclusions can be explained by the strict design principles of AUTOSAR. Namely, Classes represent the main modeling units of semantics in the AUTOSAR meta-model and the goal is to keep their complexity, coupling and cohesion as low as possible. That is why Classes usually do not have many Associations. This assures the high correlation between their growth in size and complexity, cohesion and coupling as there are not many highly coupled areas with only a few Classes and vice versa. The correlation between the growth in size and length of the Classes is implied by the existence of a well established hierarchy of Classes (e.g., Referrable and Identifiable Classes in the example in Figure 4.1) so each newly introduced Class is already a child of several other Classes.

The difference in the results of the Number of attributes metric in comparison to the other metrics is explained by the fact that Classes, as main modeling units of semantics in the AUTOSAR meta-model, may or may not contain additional descriptions in the form of Attributes (there are many Classes without Attributes, e.g., SwcToEcuMapping from Figure 4.1). This depends on the logic of Classes, not their number, so the high increase in the Number of classes does not necessarily mean the high increase in the Number of attributes.

In order to validate the accuracy of the Fan-in and Package cohesion metrics, we studied the release notes of the considered AUTOSAR meta-model releases in order to compare them to the results of these two metrics. A brief summary of the release notes is shown in Table 4.2 and the trend in the results of the Fan-in metric calculated on the 22 releases of the entire AUTOSAR meta-model is shown in Figure 4.11. The trend in the results of the Package cohesion metric is very similar due to high correlation between the results of these two metrics.
### Table 4.2: Summary of the release notes

<table>
<thead>
<tr>
<th>Release</th>
<th>Factors</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1.0</td>
<td>First release</td>
</tr>
<tr>
<td>R2.0</td>
<td>Bug-fixes only</td>
</tr>
<tr>
<td>R2.1</td>
<td>Bug-fixes, new features in the Software Component Template and the System Template packages, e.g., Measurement and calibration</td>
</tr>
<tr>
<td>R3.0.1</td>
<td>Meta-model cleanup, bug-fixes, new template BswModuleTemplate, FIBEX standard harmonization</td>
</tr>
<tr>
<td>R3.0.2  - R3.1.5</td>
<td>Bug-fixes, new concept On-Board Diagnostics in R3.1.1 (affected mostly the AR M1 layer, not the analyzed AR M2 layer)</td>
</tr>
<tr>
<td>R3.2.1</td>
<td>Bug-fixes, new concepts Partial networking, Production and development errors, End2End protection, extended Complex Device Driver</td>
</tr>
<tr>
<td>R3.2.2</td>
<td>Bug-fixes only</td>
</tr>
<tr>
<td>R4.0.1</td>
<td>Meta-model cleanup, bug-fixes, many new concepts such as Ethernet, Variant handling, Timing model, etc.</td>
</tr>
<tr>
<td>R4.0.2</td>
<td>Bug-fixes, new AUTOSAR templates StandardizationTemplate and AutosarTopLevelStructure</td>
</tr>
<tr>
<td>R4.0.3</td>
<td>Bug-fixes, new concept Partial networking</td>
</tr>
<tr>
<td>R4.1.1</td>
<td>Bug-fixes, many new concepts such as Partial networking on Ethernet, continued FIBEX harmonization and Timing model, J1939 for heavy duty vehicles, etc.</td>
</tr>
<tr>
<td>R4.1.2</td>
<td>Bug-fixes only</td>
</tr>
</tbody>
</table>

![Figure 4.11: Fan-in trend - entire AUTOSAR meta-model](image)

We can see relatively stable results of the *Fan-in* metric in releases 3.0.2 - 3.1.5 and also between releases 3.2.1 and 3.2.2 and between releases 4.1.1 and 4.1.2. This is expected as these releases contain bug-fixes only, i.e., no new features are introduced. On the other hand we can see an increase in the *Fan-in* metric results between releases 2.0 and 2.1, 4.0.2 and 4.0.3 and between releases 4.0.3 and 4.1.1 due to the introduction of new concepts. This is also expected as concepts are used to incorporate new features into the AUTOSAR meta-model. Finally we concluded that a decrease in the results of the *Fan-in* metric between releases 2.1 and 3.0.1 is related to the meta-model cleanup activity which removed the unused / obsolete elements from
the meta-model. Similar cleanup activity occurred in release 4.0.1 but due to many new concepts incorporated into this release, the results of the Fan-in metric are still increased.

4.6.5 Recommendations

Due to the logical organization of the AUTOSAR meta-model structured into different packages where each package may be used to define the properties of the models developed by separate teams, monitoring the evolution of the AUTOSAR meta-model shall be done per package bases. In this paper we presented the analysis of the top level packages of the AUTOSAR meta-model - Software Component Template and System Template - but similar analysis can be done for the packages situated lower in the hierarchy. Therefore, we recommend to the software designers of one team who plan to adopt a newer release of the AUTOSAR meta-model to analyze the changes in the relevant packages between the current and the new release following these steps:

1. Measure the complexity growth using the Fan-in metric. These results shall be used to indicate the complexity increase of the software models and tools working with the models after the adoption of the new AUTOSAR meta-model release.

2. Together with measuring the complexity, we propose to measure the increase in the Package cohesion of the relevant packages in order to estimate the work needed to be done internally inside one team.

3. In order to estimate possible integration issues between different teams and their tools, we propose to measure the increase in the Package coupling of the relevant packages as it shows the growth in communication between different packages which may be developed by separate teams.

4. Finally we propose to complement the results of these metrics (Fan-in, Package cohesion and Package coupling) by measuring the size increase of the relevant packages using the Number of classes metric. The reason for this is to assure that the metrics are in proportion (as the PCA shows) as otherwise disruptive changes may have occurred in the meta-model which may require dedicated task force to implement.

Despite the fact that we defined and analyzed the results of the assessed metrics in a case study of AUTOSAR meta-model, we believe they are applicable for quantifying the evolution of a larger set of meta-models based on MOF, e.g., the UML meta-model. This is especially the case with the domain specific industrial meta-models which are used to define the models exchanged between different parties in the development process where the distinction between the cohesion (i.e., attributes and connectors connecting the classes inside one package) and coupling properties (i.e., connectors connecting the classes in different packages) is very important. However depending on the logical structure of the analyzed meta-model, different metrics may have different significance in quantifying the meta-model evolution and also not all meta-model properties (e.g., size and complexity) may be equally affected.
4.7 Conclusion

In this paper, we assessed the applicability of 10 different metrics for quantifying the evolution of industrial meta-models with respect to 5 properties - size, length, complexity coupling and cohesion. We assessed the metrics on a case of AUTOSAR meta-model evolution at Volvo Car Corporation. The goal was to identify which of these properties are mostly affected by the evolution of the AUTOSAR meta-model and which of the assessed metrics are able to monitor them most accurately. In order to do this, we performed the Principal Component Analysis (PCA) of the results of the metrics calculated on a set of 22 releases of the AUTOSAR meta-model. We validated the chosen metrics by comparing their results with the release notes of the considered AUTOSAR meta-model releases.

We concluded that the Fan-in and Package cohesion metrics provide the most accurate results and that the Number of attributes and Cohesion ratio metrics provide the least accurate results. We also concluded that the majority of the metrics, except for the Number of attributes and Cohesion ratio, are very correlated which indicates that the evolution of the AUTOSAR meta-model is quite even for all 5 analyzed properties. Based on this, we concluded that it is enough to use only one metric for quantifying the evolution of the AUTOSAR meta-model. Due to the highest accuracy of their results, we propose to use either the Fan-in or Package cohesion metric. Finally, we made recommendations on how to combine the results of the assessed metrics to analyze the potential impact of adopting new AUTOSAR meta-model releases.

In our future work, we plan to use the metrics described in this paper to analyze the evolution of the UML 2.0 meta-model. We also plan to develop a method for estimating the effort needed to adopt a newer AUTOSAR meta-model release based on the results of the proposed metrics.
Bibliography


Chapter 5

Paper D

Identifying Optimal Sets of Standardized Architectural Features - A Method and its Automotive Application

D. Durisic, M. Staron, M. Tichy and J. Hansson

Proceedings of the 11 International ACM SIGSOFT Conference on Quality of Software Architectures, 2015
Abstract

Industrial standards are used to formalize procedures, rules and guidelines for the industry to follow. Following a standard requires continuous adoption of new standardized features where only their subset is required by individual companies. Therefore prioritization of the features and the assessment of their impact on the development projects is crucial for the success of the project. In software engineering, industrial standards are used increasingly often to standardize a language for designing architectural components of the system by defining domain-specific meta-models. The purpose is to assure the interoperability between a number of software tools exchanging the architectural models. In this paper, we present a method for identifying optimal sets of new standardized architectural features to be adopted in the development projects. The optimization is done based on the assessment of their benefit for the projects and the estimated cost of re-work in the modeling tools according to the changes in the standardized meta-model. We evaluate the method by applying it on 14 new architectural features of a new release of the AUTOSAR standard which is followed in the development of the automotive software systems.
5.1 Introduction

Industrial standards are often followed in the design and implementation of architectural components of large software systems [1]. There are two reasons for such a trend. First, the system becomes more reliable due to the use of common architectural components verified in different products. Second, the development cost and time is reduced due to increased re-usability of the architectural components and their implementations. These two reasons are especially important for distributed embedded systems - reliability due to constant increase in the size and complexity of the architectural components (see examples in the automotive domain [2, 3]) and re-usability in order to reduce high development cost related to hardware and middleware [4].

Apart from their distributed implementation, the development of large distributed embedded systems such as automotive systems is often distributed as well involving a number of actors in the development process. On one side we have OEMs (Original Equipment Manufacturers) responsible for designing and verifying the architectural components of the system. On the other side we have a chain of suppliers (e.g., application, middleware, hardware, tool suppliers) responsible for their implementation. These actors communicate by exchanging the architectural models and they may use a number of different software tools to work with them. In order to assure the interoperability between these tools, domain-specific meta-models are defined and standardized requiring a full compliance of the architectural models to their meta-models.

The design of architectural components based on the standardized meta-model requires standardization of the architectural features before their utilization in the development projects. For this reason, OEMs and their suppliers constantly incorporate new features into the standard causing thousands of changes to the standardized meta-model. Therefore it is hardly feasible for the OEMs to adopt all new features from one release of the standard in their projects. For this reason, they are usually required to make a prioritization of the architectural features in order to select their optimal subset.

The obvious question that arises is which set of features shall be adopted, i.e., how to balance the need for the new architectural features with the cost of their support in the internal and external (i.e., supplier) tools? Some features are well planned in advance and sometimes even driven by the OEM in the process of their standardization. These features are usually selected for implementation. Some features are not applicable or required by the projects and therefore rejected. However there may be other features which could have a positive impact on the final product but are not required for achieving the predefined goals. For these architectural features, a tradeoff analysis with respect to their cost and benefits shall be performed.

One of the most important factors used in the tradeoff analysis is the impact of new features on the tools used for working with the architectural models. The main reason for this is the cost of updating the tools to be able to work with the new features but also possible interoperability issues between different tools which may be caused by the changes. As these tools are based on the standardized meta-model, analyzing the evolution of the meta-model with respect to the changes imposed by the new features could be a good indicator of the potential impact of the features on the modeling tools. However due
5.2 Related work

The proposed MeFiA method shares some similarities with the ATAM (Architecture Tradeoff Analysis Method) [6] based on the CBAM (Cost-Benefit Analysis Method) [7], in particular gathering stakeholders (system architects) and discussing tradeoffs between different architectural solutions affecting the system. However we base our tradeoff analysis on the cost-benefit analysis of adopting different sets of architectural features rather than on the design of different architectural solutions.

With respect to the optimized selection of features in different product lines, Asadi et al. [8] propose a framework for automated selection of feature sets based on the functional and non-functional requirements of the system. Related to the architectural features, White et al. [9] present a method for selecting highly optimal sets of architectural features based on their resource consumption. However we are not aware of any work related to the search for optimal set of architectural features to be adopted in the development projects based on their impact on the domain-specific meta-models and software modeling tools based on them.

This papers also contributes to the area of change management of software artifacts developed in distributed working groups. The development of these
artifacts (e.g., models, specifications, code) usually relies on the existence of a change management tool containing change requests and a database with the historical versions of the artifacts. Considering the links between these two tools, Bachmann et al. [10] discuss the problems of unreported bugs and missing links in the software repository commits and propose a tool - Linkster - to automatically recover the missing links. Fischer et al. [11] analyze these links in order to find dependencies between features. Our focus is on the utilization of these links to predict the effort needed to adopt new architectural features in the development projects.

With respect to the impact of the software architecture evolution on the development projects, Gustavsson et al. [12] present the automotive study of how system architects manage architectural changes in different product lines. Eklund et al. [13] discuss the architectural concerns of extending the existing software system with new features. In the automotive domain, Dersten et al. [14] present a systematic literature review of the effects of re-factoring the AUTOSAR architecture such as lower complexity and increased efficiency. Soubra et al. [15] use functional size measurement to estimate the development effort for Electronic Control Units (ECUs)\(^1\) based on the AUTOSAR architecture. Our paper shall be considered as complementary to these studies.

5.3 Research methodology

We define our research objective according to the structure presented by Wohlin et al. [16] as:

- **Goal:** Identifying the optimal sets of standardized architectural features to be adopted in the projects.

- **Purpose:** Facilitate the decision making process of which features shall be selected.

- **View:** Project managers and system architects working on software development projects.

- **Context:** The evolution of the standardized meta-model used in the design of architectural components.

This objective is tightly related to the industrial need of car manufacturers working with architectural models based on the AUTOSAR standard as they often have to make trade-offs between adopting new standardized architectural features and the cost of their implementation. Therefore a method and a tool for automated feature impact assessment on the modeling tools used in the development which can present a set of optimal solutions can be very helpful in the decision making process of which new features to select. As the problems identified in the automotive domain served as a motivation for this research, we define our research questions focusing on the development of the automotive software systems. However we believe that other domains, such as avionics, may face similar problems.

\(^1\)Embedded system (hardware and software) responsible for one or more vehicle functions (e.g., engine control, breaking).
5.3. RESEARCH METHODOLOGY

- **Q1**: How to assess the impact of different architectural features on the tools working with models of the automotive architectural components?

- **Q2**: How to select the optimal set of features to be adopted in the automotive development projects based on their impact on different tools?

In order to provide answers to the posed research questions, we define a method - *MeFiA* - for identifying the optimal sets of architectural features to be adopted in the development projects. The method is developed in 3 steps:

1. Define how to link AUTOSAR meta-model changes to different AUTOSAR features.
   
   We conduct a case study analysis [17] of the AUTOSAR development process in order to identify means of linking the AUTOSAR meta-model changes to the AUTOSAR features. We define an approach based on the links between a change management tool (*Bugzilla*) containing the description of the features and a software code/model repository (*SVN*) containing different meta-model versions.

2. Define a measure of impact of an AUTOSAR feature on the AUTOSAR based tools.
   
   As AUTOSAR based tools are based on the AUTOSAR meta-model, quantitative analysis [18] of the changes in the AUTOSAR meta-model related to a particular feature could indicate potential re-work required in the tools to support this feature. For measuring the amount of changes between different releases of the AUTOSAR meta-model, we use the *Number of changes* (*NoC*) metric which has shown to be an effective measure of the size change [19]. We consider only changes which require certain implementation/integration effort in the AUTOSAR based tools. This is in contrast to the editorial changes (e.g., in the notes of the elements) or changes in the auxiliary parts of the AUTOSAR meta-model (e.g., *Methodology*).

3. Define how to identify the optimal sets of features to be adopted in the development projects.
   
   For identifying the optimal sets of features to be adopted in the development projects based on their cost-benefit analysis, we follow the approach based on Pareto optimality. When searching for the optimal sets of features, we consider the impact of the features included in each set on the entire AUTOSAR meta-model and also on its separate parts related to the most important roles in the automotive software development process. We used the roles and the mapping of the roles to different parts of the AUTOSAR meta-model presented in [19].

We assess the proposed method by applying it on an automotive scenario where the optimal sets of features from the AUTOSAR release 4.2.1 shall be identified. In order to extract the relevant data from the AUTOSAR meta-model, perform feature related calculations and present the optimal solutions, we developed a tool to fully automate the process.
This study represents a continuation of our work presented in [19] where we show the historical analysis of the AUTOSAR meta-model evolution and [20] where we assess a number of metrics for monitoring the evolution of the AUTOSAR meta-model.

5.4 MeFiA method definition

The goal of the MeFiA method is to identify optimal sets of new standardized architectural features to be adopted in the software development projects. In Section 5.4.1, we define a meta-data model used for calculating the Number of changes (NoC) between different meta-model versions. In Section 5.4.2, we define how to establish a link between the meta-model changes and the corresponding features. In Section 5.4.3, we show how to search for the optimal sets of features to be adopted in the projects considering their prioritization and the required implementation effort in the meta-model based tools. Finally in Section 5.4.4, we discuss our assumptions for utilization of the MeFiA method.

5.4.1 Meta-data model for the changes

In order to calculate the number of changes between different meta-model versions, we use the meta-data model presented in Figure 5.1 [19]. This meta-data model represents a simplified version of the MOF meta-model [21].

![Figure 5.1: Meta-data model used calculating NoC](image)

The meta-model is divided into Packages which contain Elements - classifiers and instances. The classifier Elements contain Attributes, Connectors of different Type (e.g., Associations, Generalizations) and Annotations describing their additional properties (e.g., regular expressions for strings). The instance Elements contain Connectors (except of Type Generalization as they represent concrete instances) and Annotations (e.g., C type, multiplicities). Finally, the Connectors and Attributes may also contain Annotations. Each
5.4. MEFIA METHOD DEFINITION

mentioned meta-element of the meta-data model may contain additional properties captured in its attributes, such as Name, Note etc.

In order to compare different meta-model versions based on the presented meta-data model, we define the Number of changes (NoC) metric which counts the total number of relevant changes (i.e., causing re-work in the AUTOSAR-based tools) between two meta-model versions [19].

We define a ‘change’ as an atomic modification, addition and removal of the meta-data model elements and their properties (e.g., Name, Note). For example if one Attribute changed both its Name and Type, this counts as two changes. Additionally when introducing or removing meta-data elements (e.g., Attributes) containing other meta-data elements (e.g., Annotations), the total number of changes considers both changes to the containing and contained meta-data elements (i.e., both Attributes and Annotations).

As an example, consider the introduction of one Attribute containing three Annotations. This counts as four changes - one for the Attribute and three for the Annotations. This way of calculating the total number of changes is justified by the fact that introduction of one Element cannot be counted as one change, like for example the introduction of one Annotation, as it requires higher implementation effort in the meta-model based tools.

To identify meta-elements in different meta-model versions, we used their UUIDs (Universally Unique IDentifiers of the objects) except for the Annotations where we used their Name as it is able to uniquely identify them in the context of one meta-data element.

5.4.2 Linking meta-model changes to features

In order to link meta-model changes to specific architectural features, certain process for implementing the changes in the meta-model needs to be established. This is especially the case with standardized features where many different parties may be involved in the definition of the final solution. The process we utilize in this paper relies on the existence of two commonly used tools in the distributed software development - a change management tool (e.g., Bugzilla, Jira) and a software repository (e.g., SVN, Git).

Change management tools such as Bugzilla can be used for documentation and traceability of the new standardized architectural features incorporated into different releases of the standard. Software repositories on the other hand can be used for documentation and traceability of different versions of the architectural meta-models between different releases of the standard. In order to link meta-model changes to features, a link between these two tools can be established by following a process depicted in Figure 5.2.

For each new feature to be implemented in the meta-model, an entry in the change management tool shall be created with a unique identifier (1). Software designer implementing the changes in the standardized meta-model shall use this identifier (2) in the commit message when committing a new version of the meta-model to the software repository (3). The process of modifying the meta-model and committing a new version to the software repository related to the same feature can be repetitive. Every time a commit to the repository is made, a comment is added to the bug with the identifier from the commit message (4). To assure that no links are omitted by the change implementers,
the software repository should be configured to accept only certain structure of commit messages, e.g., a regular expression starting with the unique identifier of the entry in the change management tool (e.g., #12345).

### 5.4.3 Optimizing the set of adopted features

The search for the optimal sets of new standardized features to be adopted in the development projects is a multi-objective optimization problem [22, 23] with two objectives:

1. Maximize the weighted number of features to be adopted based on their priority/weight.
2. Minimize the effort needed to implement the changes in the meta-model based tools based on the number of changes in the meta-model.

As the number of new features in different releases of the standard are limited to a reasonably high number and also considering the fact that the execution time is not critical, exhaustive algorithm which considers all possible combinations of features (i.e., starting with feature 1 only and ending with all features) is the most suitable algorithm.

For representing different solutions, a bit string $s$ of length equal to the total number of new features $n$ can be used as shown in formula 5.1.

$$s = (s_1, s_2, ..., s_n) \quad (5.1)$$

Each bit $s_i$ in the bit string corresponds to one feature $f_i$ ($s_1$ to feature $f_1$, etc.) and the value of the bit indicates whether the corresponding feature
is included in the solution (value 1) or not (value 0). The solutions represent all possible combinations of bit values in the bit string which yields $2^n$ different solutions.

In order to calculate the weighted number of features for each solution, a weight factor on an interval scale of 1 to 5 shall be assigned by the system designer to each feature where 5 is considered as the most important. Then the total $wNoF$ for one solution represents the sum of the weights $w_i$ of all features included in this solution, as shown in formula 5.2.

$$wNoF(s) = \sum_{i=1}^{n} w_i \times s_i \quad (5.2)$$

In order to measure the effort needed to adopt the features from one solution, we calculate the total $NoC$ for this solution as the sum of the $NoC$ for each feature included in the solution, as shown in formula 5.3.

$$NoC(s) = \sum_{i=1}^{n} NoC(f_i) \times s_i \quad (5.3)$$

This formula is based on the assumption that each change in the meta-model requires certain implementation effort in the meta-model based tools. Therefore the more changes we have, the more effort is needed to adopt the features causing these changes.

In order to represent all possible solutions and identify the optimal ones with respect to their $wNoF$ (objective 1) and $NoC$ (objective 2), a Pareto optimality chart presented in Figure 5.3 can be used.

![Pareto optimality chart](image)

Figure 5.3: Pareto optimality chart

On the $x$ axis we present the $NoC$ and on the $y$ axis we present the $wNoF$. Then the solutions lying on the top and left most part of the chart form a

---

$^2$We consider a scale of 1-5 to be the optimal but different scales could be more suitable for other situations.
Pareto front. These solutions are considered as better solutions, with respect to the two objectives, than the solutions lying below and to the right of them. For example solution \( b \) has both higher \( wNoF \) and lower \( NoC \) than solutions \( e, f, g, i \) and \( j \). Therefore the optimal solution shall be discussed among the solutions lying on the Pareto front. This may significantly reduce potentially high number of solution and as such facilitate the decision making process.

Additionally, one solution shall be excluded from the consideration in one of the following cases:

1. If there is a dependency between two or more features (i.e., one feature cannot be implemented without another feature) and a solution contains only a subset of the dependent features.

2. In case there is no need for some features and a solution contains at least one of them.

3. In case some features are required to be implemented and a solution does not contain all of them.

5.4.4 Assumptions for the MeFiA method

The MeFiA method is designed and assessed in the automotive domain where software development is done following the AUTOSAR standard. However we think that the applicability of this method could be extended to other domains developing software architectures based on a standard such as avionics based on IMA (Integrated Modular Avionics) [24], or banking based on BIAN (Banking Industry Architecture Network) [25]. In particular, the MeFiA method could be valuable for companies where:

1. The development of architectural models is done based on a standardized meta-model which defines the syntax and semantics for the models and serves as a basis for the development of the modeling tools.

2. The architectural models are exchanged between a number of actors involved in the development process.

3. A number of actors is involved in the development of the standardized meta-model.

The first point implies that the adoption of new meta-model versions is needed to enable innovations in the development projects. The second point implies that the adoption of new meta-model versions may potentially cause interoperability issues between modeling tools used by different actors in the development process. The third point implies that possibly many new features driven by different actors may be incorporated into the new versions of the standardized meta-model which requires careful cost-benefit analysis of which new features shall be selected for implementation.

In order to utilize the MeFiA method in other domains, the development of the standardized meta-model needs to satisfy the following conditions:

1. Standardized features shall be stored in a change management system, e.g., defect management system such as Bugzilla or Jira.
2. Different meta-model version shall be stored in a software repository such as SVN or Git and each meta-model commit shall be linked to the corresponding entry in the change management system.

3. Changes related to different features should not be committed simultaneously as the proposed method is not able to automatically detect which changes are related to which features. If there are cases like this, changes in these commits shall be analyzed by system architects and designers and assigned to the right feature.

Additionally, the $NoC$ metric is based on the assumption that all changes have equal weight, i.e., they all require similar implementation effort in the meta-model based tools. However changes introducing/breaking dependencies may require more effort. This could be improved in future by classifying the changes into different types, which can be automatically detected, and assigning a weight to each type.

5.5 Automotive software development

This section describes the development of the automotive software systems following the AUTOSAR standard which we used for the evaluation of the MeFiA method.

The development of the automotive software systems is distributed as they are developed in a collaborative environment which involves a number of actors. On one side we have car manufacturers (OEMs) responsible for designing and verifying the architecture of the system. On the other side we have several layers of suppliers (e.g., application software suppliers, tool vendors, hardware suppliers) responsible for design, implementation and verification of the specific architectural components of the system [26]. As each actor in the development process may use its own tools for working with the architectural models, the exchange of models between the actors is quite challenging.

In order to facilitate this distributed development of the automotive software systems, AUTOSAR standard was introduced [27] as a joint partnership of the OEMs and their suppliers on the European market and wider. One of the main goals of AUTOSAR is to clearly separate the responsibility of different actors in the development process. For this purpose, a 3-layer software architecture [28] has been developed where the application software (i.e., vehicle functions such as auto-braking when pedestrians are detected in front of the car) is clearly separated from the underlying basic software (e.g., communication between ECUs, diagnostic services, etc.) and hardware [29].

Based on this architecture, AUTOSAR standardizes the exchange format for the architectural models. This is done by defining a meta-model which specifies the syntax and semantics of the automotive modeling environment [30, 31] and serves as a basis for the development of the AUTOSAR based tools used for modeling the architectural components (e.g., application software components or basic software modules) of the system. In order to assure the interoperability between different AUTOSAR based tools, the exchanged architectural models needs to be fully compliant to the AUTOSAR meta-model. Figure 5.4 shows a simplified example of the usage of the AUTOSAR meta-model to allocate software components onto different ECUs.
The meta-model to the left defines how to allocate software components onto ECUs while the model to the right instantiates this meta-model by mapping the actual EnginePowerUnit software components onto EngineControlModule ECU. Software components and ECUs represents one of the main architectural units of the automotive software system and their allocation onto ECUs is an architectural feature. Another example of the architectural feature may be the use of Ethernet electronic bus as a communication medium between two or more ECUs.

The architectural models are usually expressed in XML [32] and they are delivered by OEMs to their suppliers to continue with the implementation of the software, e.g., by developing behavioral models in tools such as Matlab Simulink. Before importing the models into the AUTOSAR based tools, they are validated by the AUTOSAR XML schema [33, 34] which is generated from the AUTOSAR meta-model (see [35] for more details about generating XML schema from the UML model). This process is depicted in Figure 5.5.
To track the changes to the AUTOSAR specifications including the AUTOSAR meta-model, Bugzilla tool is used. Each change can be classified as clarification, correction or a new feature. For the new features which influence several different parts of the AUTOSAR architecture, new concepts are created and elaborated by different experts. The incorporation of the new concepts into the AUTOSAR releases is also documented in Bugzilla where a separate implementation task is created for the AUTOSAR meta-model.

For the development of the AUTOSAR specifications and meta-model, SVN repository is used. When updating the meta-model, the corresponding implementation task from the AUTOSAR Bugzilla needs to be referenced at the beginning of the SVN commit message (e.g., #54321). This reference is required by the SVN in order to avoid situations where SVN commits are not linked to any Bugzilla entries. This enables full traceability of the changes to the AUTOSAR meta-model and other specifications and Bugzilla entries where the requests for these changes are described and elaborated.

5.6 Applying MeFiA on AUTOSAR features

We apply the MeFiA method on a set of 14 new features (referred to as concepts in AUTOSAR) incorporated into the AUTOSAR release 4.2.1. In order to fully automate the generation of the optimal solutions, we implemented a tool which is used for this purpose at Volvo Cars. A brief description of each feature is presented below (the features have no dependencies between each other as stated in the feature documents of AUTOSAR):

- **Feat 1: Ethernet Switch Configuration** - provides means to configure Ethernet switches in an ECU.

- **Feat 2: Sender-Receiver Serialization** - significantly reduces the number of signals needed for transmission of complex data.

- **Feat 3: CAN FD** - introduces a new communication protocol for CAN bus with higher bandwidth.

- **Feat 4: Efficient COM for Large Data** - faster transmission of large data through the ECU.

- **Feat 5: End-to-End Extension** - extends the safety communication means between ECUs for transmission of large data via TCP/IP.

- **Feat 6: Global Time Synchronization** - provides a common time base for accurate ECU data correlation.

- **Feat 7: Support for Post-Build ECU Configuration** - enables the configuration of ECU variants in one vehicle and different car lines.

- **Feat 8: Secure On-Board Communication** - provides mechanisms for securing the communication between vehicle and the outside world.

- **Feat 9: Safety Extensions** - provides mechanisms to realize and document functional safety of AUTOSAR systems (e.g., according to the ISO 26262).
• **Feat 10: Decentralized Configuration** - provides means for transferring diagnostic needs of OEMs to suppliers.

• **Feat 11: Integration of Non-AUTOSAR Systems** - enables integration of non-AUTOSAR (e.g., Genivi) systems into AUTOSAR during development.

• **Feat 12: Efficient Non-Volatile Data Handling** - provides efficient mechanisms for software components to handle non-volatile data.

• **Feat 13: ECU State Manager Enhancement for Multi-Core** - provides support for state handling on multi-core ECUs.

• **Feat 14: ASIL-QM protection** - provides means to protect modules developed according to safety regulations from other modules.

In order to calculate the number of changes caused by each feature, we analyzed the changes between 97 SVN commits of the AUTOSAR meta-model. Out of these commits, 80 referred to only one feature and 17 additionally referred to other implementation tasks (13 to defect corrections and 4 to other features). We excluded the changes not related to the analyzed feature from the latter 17 commits by analyzing them together with the AUTOSAR team at Volvo Cars.

As different companies may be interested in different features where some are required to be implemented, some are not required and some may be considered only in case the cost of their implementation is acceptable, different scenarios for the usage of the MeFiA method are possible. For the purpose of this paper, we define the following scenario: A company wants to implement **Feat 1 (Ethernet Switch Configuration)** and **Feat 3 (CanFD)** and all other features are subject to cost-benefit analysis with equal weights.

Section 5.6.1 discusses the optimal sets of features for the given scenario based on the analysis of their impact on the entire AUTOSAR meta-model. As the evolution of industrial meta-models may have significantly different impact on different parts of the AUTOSAR meta-model affecting different roles (teams) [19], the impact of the changes on different roles shall also be considered. This is especially important when analyzing the impact of feature related changes as some features may be related only to a limited number of roles. Section 5.6.2 shows how to search for the optimized set of features considering their impacts on a particular role and Section 5.6.3 presents an example of how to aggregate the results of the analysis for different roles and use them in the decision making process.

### 5.6.1 Optimization for the entire meta-model

Before searching for the optimal sets of features to be adopted in the development projects, it should be analyzed whether there are features which do not affect the AUTOSAR meta-model or features causing significantly more meta-model changes than other features. The adoption of these outliers should be analyzed separately in order not to cause major diss-balance in the results.

---

3For simplicity and having in mind that different companies may use different prioritization of features, we assigned weight ‘1’ to all analyzed features.
In order to identify these kinds of features, a chart (e.g., histogram) presenting the total number of meta-model changes for each feature can be used. Figure 5.6 shows the results of the \( \text{NoC} \) metric calculated for the 14 new features of the AUTOSAR release 4.2.1.

![NoC per feature chart](chart)

**Figure 5.6: NoC per feature**

We can see that the results of the \( \text{NoC} \) metric are very diverse ranging from 0 in case of \textit{Feat 11 (Integration of Non-AUTOSAR Systems)}\(^4\) to 17961 in case of \textit{Feat 7 (Support for Post-Build ECU Configuration)}. These two features represent the outliers so we decided to exclude them from the analysis of optimal features with the recommendation to do the cost-benefit analysis for their adoption separately (e.g., \textit{Feat 7} should probably be selected only if it is absolutely required due to its high number of changes affecting the AUTOSAR meta-model).

Additionally based on the scenario presented above, \textit{Feat 1 (Ethernet Switch Configuration)} and \textit{Feat 3 (CanFD)} shall be added to all solution as they are required to be adopted. Figure 5.7 shows the Pareto optimality chart with Pareto front containing 11 optimal solutions (\textit{s1} - \textit{s11}) based on their impact on the entire AUTOSAR meta-model. Table 5.1 shows which features are included in which solution.

We can see that a higher increase in the number of changes needed to be implemented to support an additional feature starts with solutions \textit{s7} and \textit{s8}. Therefore the optimal solution should be searched among solutions \textit{s6} and \textit{s7}, i.e., \textit{Feat 1 (Ethernet Switch Configuration)}, \textit{Feat 3 (CAN FD)}, \textit{Feat 9 (Safety Extensions)}, \textit{Feat 14 (ASIL-QM protection)}, \textit{Feat 12 (Efficient Non-Volatile Data Handling)}, \textit{Feat 13 (ECU State Manager Enhancement for Multi-Core)}, \textit{Feat 5 (End-to-End Extension)} and alternatively \textit{Feat 4 Efficient COM for Large Data} shall be selected for adoption. However before making a final decision, the impact analysis of these features on different roles shall be performed first, at least for the most critical roles.

\(^{4}\)\textit{Feat 11} brings no changes because it does not affect the AUTOSAR based tools.
5.6.2 Role-based optimization

For the role based analysis of the AUTOSAR meta-model changes, we consider 7 major roles in the AUTOSAR based automotive software development process which we defined in [19] (a mapping of roles to different parts of the AUTOSAR meta-model can also be found in this paper). A brief summery of these roles is presented below:

- **Role 1: Application software designers** - a team at the OEMs responsible for designing vehicle functions by defining software components and their interaction (e.g., data exchange points between components).

- **Role 2: ECU communication designers** - a team at the OEMs responsible for designing the communication between ECUs (e.g., creation of signals and their transmitting on the electronic buses).

- **Role 3: ECU basic software configurators** - a team at the OEMs responsible for specifying different basic software configuration possibilities (e.g., setting configuration parameter values after building the ECU software).
• **Role 4: Basic software designers** - a team at the basic software suppliers responsible for designing the basic software modules (e.g., interfaces between the modules, supported services, etc.).

• **Role 5: ECU communication configurators** - a team at the application software suppliers responsible for configuring ECU communication related basic software modules.

• **Role 6: Diagnostics configurators** - a team at the application software suppliers responsible for configuring diagnostics related basic software modules.

• **Role 7: Upstream mapping tool developers** - a team at the tool suppliers responsible for deriving parts of the ECU configuration (i.e., parameter values) from the system models (“upstream mapping”).

In order to identify the most critical roles, we analyze the chart presented in Figure 5.8 which shows the number of feature related changes affecting each role separately (note that there may be other non-feature related changes affecting these roles such as corrections of defects).

![Concept changes per role](image)

Figure 5.8: Number of feature changes per role

This conforms to our conclusion in paper [19] that the evolution of the AUTOSAR meta-model mostly affects **Role 5 (ECU communication configurators)** and **Role 6 (Diagnostics configurators)** while other roles are less affected. Low impact is especially important for **Role 1 (Application software designers)** and **Role 2 (ECU communication designers)** as the architectural models developed by these roles are usually exchanged between OEMs and suppliers and therefore they affect multiple actors in the development process. Considering this and the fact that the role of **ECU communication designers** is around 5 times more affected by the changes than the role of **Application software designers**, we consider the **ECU communication designers** as the most critical role.

Based on the scenario presented above, Figure 5.9 shows the Pareto optimality chart with Pareto front containing 9 optimal solutions (s12 - s20) based on their impact on the **ECU communication designers** role. Table 5.2 shows which features are included in which solution.
Figure 5.9: Optimal sets of features (*ECU communication designers*)

Table 5.2: Features of the optimal solutions (*ECU communication designers*)

<table>
<thead>
<tr>
<th>Solutions</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>s12</td>
<td>Feat 1, 3, 13, 14</td>
</tr>
<tr>
<td>s13</td>
<td>Feat 1, 3, 13, 14, 4</td>
</tr>
<tr>
<td>s14</td>
<td>Feat 1, 3, 13, 14, 4, 9</td>
</tr>
<tr>
<td>s15</td>
<td>Feat 1, 3, 13, 14, 4, 9, 12</td>
</tr>
<tr>
<td>s16</td>
<td>Feat 1, 3, 13, 14, 4, 9, 12, 8</td>
</tr>
<tr>
<td>s17</td>
<td>Feat 1, 3, 13, 14, 4, 9, 12, 8, 5</td>
</tr>
<tr>
<td>s18</td>
<td>Feat 1, 3, 13, 14, 4, 9, 12, 8, 5, 10</td>
</tr>
<tr>
<td>s19</td>
<td>Feat 1, 3, 13, 14, 4, 9, 12, 8, 5, 10, 6</td>
</tr>
<tr>
<td>s20</td>
<td>Feat 1, 3, 13, 14, 4, 9, 12, 8, 5, 10, 6, 2</td>
</tr>
</tbody>
</table>

If we compare solutions *s12* and *s15*, we can see that with a relatively small increase in the number of changes we can implement 3 additional features - *Feat 4 (Efficient COM for Large Data)*, *Feat 9 (Safety Extensions)* and *Feat 12 (Efficient Non-Volatile Data Handling)*. We can also see that a big increase in the number of changes required to be implemented for adopting an additional feature starts with solution *s17*, in particular with *Feat 5 (End-to-End Extension)*. Therefore the optimal solution should be searched among solutions *s15* and *s16*.

5.6.3 Aggregated role-based optimization

In the previous two sub-sections, we discussed several optimal solutions based on the impact of their features on the entire meta-model (*s6* and *s7*) and the *ECU communication designers* role which is considered as the most critical (*s15* and *s16*). This means that apart from the required *Feat 1 (Ethernet Switch Configuration)* and *Feat 3 (CAN FD)*, the decision about the adoption of *Feat 4 (Efficient COM for Large Data)*, *Feat 5 (End-to-End Extension)*, *Feat 8 (Secure On-Board Communication)*, *Feat 9 (Safety Extensions)*, *Feat 12 (Efficient Non-Volatile Data Handling)*, *Feat 13 (ECU State Manager Enhancement for Multi-Core)* and *Feat 14 (ASIL-QM protection)* shall be made.
Before making a final decision about the adoption of these features, their impact on the other roles in the development process shall be considered. Figure 5.10 shows the number of changes needed to be implemented by different roles to support each one of the considered features.

<table>
<thead>
<tr>
<th>NoC</th>
<th>Role 1</th>
<th>Role 2</th>
<th>Role 3</th>
<th>Role 4</th>
<th>Role 5</th>
<th>Role 6</th>
<th>Role 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feat 4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>214</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Feat 5</td>
<td>1</td>
<td>27</td>
<td>2</td>
<td>0</td>
<td>32</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Feat 8</td>
<td>0</td>
<td>27</td>
<td>0</td>
<td>0</td>
<td>525</td>
<td>0</td>
<td>19</td>
</tr>
<tr>
<td>Feat 9</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Feat 12</td>
<td>16</td>
<td>22</td>
<td>8</td>
<td>8</td>
<td>64</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>Feat 13</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>84</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Feat 14</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 5.10: The impact of the features on other roles

We can see that the impact of these features on the other roles is smaller than their impact on the analyzed ECU communication designers role except for the ECU communication configurators role. Therefore the decision about which new features shall be adopted together with Feat 1 (Ethernet Switch Configuration) and Feat 3 (CAN FD) shall be made based on the assessment of their impact on the ECU communication configurators role.

5.7 Conclusion and future work

Following a standard in the development of software architectures requires continuous adoption of new standardized features where only their subset is required by individual companies. In order to decide upon which set of new features shall be adopted in the development projects, the assessment of their impact on the software tools used for modeling the architectural components is an important aspect in the decision making process. This is specially the case with large distributed systems where the architectural models are exchanged between a number of actors in the development process in order to assure the interoperability between different tools working with the models.

In order to facilitate the decision making process of which new standardized architectural features shall be adopted in the development projects, we defined a method - MeFiA - for identifying the optimal sets of features. These optimal sets are identified based on the tradeoff analysis between their importance for the projects and the amount of re-work needed in the software modeling tools caused by the changes in the standardized meta-model.

We evaluate the MeFiA method in the automotive domain where software architectures are developed following the AUTOSAR standard. We present the cost-benefit analysis for adoption of 14 new architectural features of the AUTOSAR release 4.2.1. The analysis is done based on a scenario considering different roles in the development process.

We concluded that the proposed method is applicable for identifying the optimal sets of new architectural features to be adapted in the automotive development projects. However we also believe that the method is applicable to a wider range of domains where software is developed based on industrial,
domain-specific meta-models and where the link between a change management tool and a software repository is maintained. Further empirical studies are needed to support this.

In our future work we intend to evaluate the robustness of the MeFiA method to violations of the method assumptions, in particular the assumption of equal weights for all meta-model changes. We plan to address this by classifying the changes into different categories where each category contains a predefined weight. We also plan to investigate the impact of different prioritization of features in a case study at different companies.
Bibliography


Chapter 6

Paper E

ARCA - Automated Analysis of AUTOSAR Meta-Model Changes

D. Durisic, M. Staron, M. Tichy and J. Hansson

Proceedings of the 7th International Workshop on Modelling in Software Engineering, 2015
Abstract

Software architecture of automotive software systems on the European market and wider is designed following the AUTOSAR standard. This requires continuous adoption of new AUTOSAR releases in the development projects in order to enable new innovative solutions in cars. Under these circumstances, the analysis of impact of the AUTOSAR meta-model changes on the modeling tools used in the development is crucial for avoiding delays and increased cost. However due to tens of new features combined with thousands of meta-model changes between consecutive releases of AUTOSAR, tool support is needed for such analysis. In this paper we present a systematic method and a tool - ARCA - for automated analysis of the AUTOSAR meta-model changes. The tool is able to identify relevant changes affecting the modeling tools used by different roles in the development process and present the optimal set of new features to be adopted in the projects. The goal of the tool is to enable faster and cheaper software innovation cycles in cars.
6.1 Introduction

The development of automotive software systems and their architectures on the European market and wider is mostly based on the AUTOSAR [1] (AUTomotive Open System ARchitecture) standard [2]. One reason is the possibility to re-use existing architectural components and their implementations (e.g., related to middleware and hardware [3]) but also to more easily exchange the architectural models between the modeling tools of different software vendors. In order to facilitate the exchange of these models, AUTOSAR defines a meta-model and requires full compliance of the models to the AUTOSAR meta-model. We consider a model as an abstract representation of a software system and a meta-model as a model which defines the syntax and semantics of a particular domain-specific modeling environment [4, 5].

The development based on the standardized meta-model requires constant adoption of new meta-model releases in the development in order to enable new innovative solutions in car projects. A good example of such a solution is Ethernet as a communication medium between different Electronic Control Units (ECUs) responsible for one or more vehicle functions (e.g., engine control) in a distributed automotive system. However as new releases of AUTOSAR usually bring thousands of changes to the AUTOSAR meta-model (e.g., more than 33 000 changes between two consecutive releases 4.1.3 and 4.2.1 as shown later), careful analysis of the impact of these changes on different modeling tools supporting these solutions is required before their implementation [6].

In particular, automotive software designers are often confronted with decisions about which newer AUTOSAR release or subsets of its new features to adopt in the development projects. They also want to know which roles in the development process will be mostly affected by the changes. Due to the constant increase in the size and complexity of the AUTOSAR meta-model related to new car functionalities [7] (e.g., R4.2.1 is 4-5 times more complex than R3.2.3 for different roles as shown later), tool support is needed to quickly identify relevant changes for the most critical roles and facilitate the cost-benefit analysis of adopting different features.

In this paper, we present a systematic method and a tool - ARCA (Autosar Change Analyzer) - for automated analysis of AUTOSAR meta-model changes affecting modeling tools used by a set of defined roles. The tool has 3 main functionalities:

1. Quantifying and presenting the changes between different versions of the AUTOSAR meta-model.

2. Presenting the results of a number of software metrics characterizing the AUTOSAR meta-model evolution.

3. Quantifying and presenting the changes caused by specific features of a new AUTOSAR release.

Based on the first two functionalities, a decision about the adoption of a specific new AUTOSAR release can be made. This includes the identification of the most critical roles affected by the changes. We studied in [6] the evolution of the AUTOSAR meta-model for different roles and assessed the
6.2 AUTOSAR BASED SOFTWARE DEVELOPMENT

The development of the automotive software systems is distributed as they are developed in a collaborative environment which involves a number of actors. On one side we have car manufacturers (OEMs - Original Equipment Manufacturers) responsible for designing and verifying the architecture of the system. On the other side we have different layers of suppliers (e.g., application software suppliers, tool suppliers, hardware suppliers) responsible for design, implementation and verification of the specific architectural components [9]. As each party in the development process may use their own tools for working with the architectural models, the exchange of these models between different actors is quite challenging.

In order to facilitate this distributed development, AUTOSAR standard was introduced [10] as a partnership of OEMs and their suppliers. One of the main goals of AUTOSAR is to standardize the exchange format for the architectural models of the system. This is done by defining a meta-model which specifies the syntax and semantics of the automotive modeling environment [4, 5] and serves as a basis for the development of the modeling tools.

Figure 6.1 shows a simplified example of the usage of the AUTOSAR meta-model to allocate software components onto ECUs.

---

**Figure 6.1:** AUTOSAR Meta-Model example
The meta-model to the left defines how to map software components to ECU s while the model to the right instantiates this meta-model by mapping actual EnginePowerUnit software components to EngineControlModule ECU.

The development of the AUTOSAR meta-model is done with the help of two tools - a change management tool Bugzilla and an SVN repository. For implementing changes related to new features, the process depicted in Figure 6.2 is followed.

![Figure 6.2: Linking meta-model changes to AUTOSAR features](image)

For each new feature to be implemented in the standard, an entry in Bugzilla is created with a unique identifier (1) which contains the description of the feature and the agreed solution. Software designers implementing the changes in the AUTOSAR meta-model use this identifier (2) in the commit message when committing the new version of the meta-model to the SVN (3). Several SVN commits of the meta-model may be related to one feature. Every time a commit is made, a comment is added to the Bugzilla entry with the identifier from the commit message (4). To assure that no links are omitted by the change implementers, the SVN repository shall be configured to accept only certain structure of commit messages, e.g., a regular expression starting with the unique identifier of the Bugzilla entry (e.g., #12345).

### 6.3 Related work

A number of studies focus on the coupled evolution of models and meta-models and assessing the impact of meta-model changes on different artifacts. For example Ruscio et al. [11] address the impact of meta-model evolution on the entire meta-modeling eco-system, e.g., models, transformations and modeling tools and Mendez et al. [12] show how to perform the automated transformation of models according to the meta-model changes. Our paper contributes to these studies by analyzing the impact of meta-model changes on the modeling tools with the focus on supporting new features.

A number of software tools exist today for supporting the analysis of repository changes for different software artifacts such as VCS-Analyzer presented...
by Fontana et al. [13] or the change management tool presented by Li et al. [14]. However most of these tools are not tailored to the analysis of meta-model evolution and they are not capable of linking meta-model changes to different system features.

Additionally Poncin et al. present FLASR - a framework for analyzing software repositories by combining different repositories and matching related software development events [15]. We utilize a part of this framework (SVN-Bugzilla links) for linking AUTOSAR meta-model changes to features.

### 6.4 ARCA tool

The description of the ARCA tool is organized in 5 sub-sections. The first one contains a description of the tool’s architecture while the latter 3 contain the definition of the main functionalities supported by the tool. The last section shows an example of how to combine all functionalities in car projects.

#### 6.4.1 The architecture of ARCA tool

For the analysis of changes between different versions of the AUTOSAR meta-model, the ARCA tool uses a meta-data model presented in Figure 6.3 which represents a simplified version of the MOF meta-model [16].

![Figure 6.3: Meta-data model](image)

We define a change as an atomic modification, addition or removal of the meta-data model elements and their properties (e.g., Name, Note). For example if an Attribute changed both its Name and Type, this represents two changes. Additionally when introducing or removing meta-data elements (e.g., Attributes) containing other meta-data elements (e.g., Annotations), both changes to the containing and contained meta-data elements are considered. The comparison is done based on the unique element identifiers in the Enterprise Architect model which contains a particular version of the AUTOSAR meta-model.
Data-models instantiating this meta-data model (.mod files) are obtained by extracting the data from the Enterprise Architect (.eap) files corresponding to a specific AUTOSAR meta-model version (see 1 in Figure 6.4). Two data-models need to be loaded into the tool (see 2 in Figure 6.4) before further analysis of the changes is possible. The reason why the tool works with its own data-models is to increase the performance, i.e., it is more than 100 times faster to load the extracted data-model in comparison to querying the Enterprise Architect file.

Before using the ARCA tool for different types of change analysis, two important aspects need to be configured: (i) which types of changes shall be considered and (ii) which roles shall be used in the analysis. The first aspect is important as certain changes may not require any implementation effort in the AUTOSAR meta-model based tools, e.g., changes in the format of the unique identifiers of the meta-elements or in their notes. The second aspect is important as the development of automotive software systems involves a number of actors so the role based analysis of the changes may indicate which teams (i.e., their modeling tools) will be mostly affected.

The configuration of these two aspects is done by importing the configuration file (.xml) before the analysis (see 5 in Figure 6.4). Regarding the considered types of changes (referred to as "relevant changes"), it is possible to specify which meta-model packages shall be excluded from the analysis. It
is also possible to specify whether the changes in the notes of the elements shall be considered and which annotations shall be excluded. Consideration of the relevant changes only can be enabled and disabled via the ‘Relevant only’ check-box (see 3 in Figure 6.4).

As the AUTOSAR meta-model is organized in logical packages, roles in the configuration file are defined as collections of packages affecting modeling tools used by the corresponding role. We defined in [6] 7 major roles in the automotive software development process but new roles can be defined.

The link between meta-model changes and AUTOSAR features is established by analyzing the AUTOSAR meta-model changes between SVN commits referring to the Bugzilla entry which corresponds to the analyzed feature.

6.4.2 Quantifying/presenting the meta-model changes

This functionality enables a comparison between two AUTOSAR meta-model versions (e.g., two AUTOSAR meta-model releases) for a selected role (see 7 in Figure 6.4) after the corresponding .mod files have been loaded. By selecting the ‘Metrics’ check-box in the ‘Show changes’ panel (see 9 in Figure 6.4), we can see the total number of changes between the chosen meta-model versions together with the number of modified, added and removed elements, attributes and packages (see 11 in Figure 6.4). In addition to the quantitative analysis of the changes, it is possible to list all changes and modified, added and removed elements, attributes and packages by selecting the corresponding check-boxes in the ‘Show changes’ panel (see 12 in Figure 6.4).

There are two main use-cases for this functionality:

1. Analyzing the impact of switching from one AUTOSAR release to another on the tools used by different roles in the development process.

2. Constant follow-up of the changes in the AUTOSAR meta-model between two releases in order to influence their standardization (e.g., prevent the removal of an element which is planned to be used in future).

Apart from the analysis of changes between two AUTOSAR meta-model versions, the ARCA tool can be used for generating .csv reports from the comparison of multiple meta-model versions specified in the configuration file (i.e., paths to the corresponding .mod files are provided) for all defined roles (see ‘Report changes’ under 6 in Figure 6.4). Based on these reports, heat-maps showing the number of changes (or the number of modified, added and removed elements, attributes and packages) which need to be implemented in the AUTOSAR based tools to switch from one AUTOSAR release to another can be created for each role.

The main use-case for the heat-maps is during the decision making process of which new AUTOSAR release shall be adopted in the development process. The reason for this is that they can serve as a good initial indicator of potential cost needed to update the AUTOSAR based modeling tools used by different roles. An example of such a heat-map containing the number of relevant changes for the AUTOSAR releases 3.1.1 - 4.2.1 considering their impact on the entire meta-model is shown in Figure 6.5.
We can see that the majority of changes are needed when switching from a late release in branch 3.2 to a late release in branch 4.2 as these two branches have been developed in parallel for some time leading to their divergence.

6.4.3 Presenting the results of software metrics

This functionality enables a comparison between results of a number of software metrics applied on the chosen two AUTOSAR meta-model versions for a selected role (see 7 in Figure 6.4) after the corresponding .mod files have been loaded. However this time the presentation of the results (see 10 in Figure 6.4) depends on the selected check-boxes of the 'Show metrics' panel (see 8 in Figure 6.4). The metrics are divided into 5 categories according to the properties defined by Briand et al. [17] which include the following metrics:

- **Size**: Number of elements, Number of attributes
- **Length**: Dept of inheritance
- **Complex**: Fan-in, Fan-out and Fan-IO (Fan-in + Fan-out)
- **Coupling**: Coupling between obj. and Package coupling
- **Cohesion**: Package cohesion and Cohesion ration

While the size and length metrics are based on the commonly used UML metrics [18], the complexity, coupling and cohesion metrics are based on the interaction between different elements (meta-classes) in the meta-model, i.e., based on Connectors (associations). Even though in the modeling world associations can be considered as attributes of the classes, in case of industrial meta-models they may have slightly different semantic as meta-classes there represent logical entities whose instances may be modeled by separate teams. Therefore the introduction / removal of one association may have a wider impact than the introduction / removal of one attribute which describes only one meta-class. For this reason, we analyzed the Connectors in the context of complexity, coupling and cohesion rather than in the context of size. A detailed definition of all presented metrics can be found in [8].

The main use cases for this functionality is to analyze the impact of the AUTOSAR meta-model changes on different roles and their interaction (model

<table>
<thead>
<tr>
<th>NoC</th>
<th>3.1.1</th>
<th>3.1.2</th>
<th>3.1.3</th>
<th>3.1.4</th>
<th>3.1.5</th>
<th>3.2.1</th>
<th>3.2.2</th>
<th>3.2.3</th>
<th>4.0.1</th>
<th>4.0.2</th>
<th>4.0.3</th>
<th>4.1.1</th>
<th>4.1.2</th>
<th>4.1.3</th>
<th>4.2.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3.1251</td>
<td>3.1545</td>
<td>3.1650</td>
<td>3.1722</td>
<td>5.2322</td>
<td>5.3718</td>
<td>5.5827</td>
</tr>
<tr>
<td>3.1.2</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>1139</td>
<td>8349</td>
<td>9188</td>
<td>31265</td>
<td>35499</td>
<td>3.1545</td>
<td>3.1650</td>
<td>3.1722</td>
<td>3.1722</td>
<td>5.2322</td>
<td>5.3718</td>
<td>5.5827</td>
</tr>
<tr>
<td>3.1.3</td>
<td>0</td>
<td>274</td>
<td>359</td>
<td>7084</td>
<td>8349</td>
<td>9188</td>
<td>31265</td>
<td>35499</td>
<td>35499</td>
<td>3.1650</td>
<td>3.1722</td>
<td>3.1722</td>
<td>5.2322</td>
<td>5.3718</td>
<td>5.5827</td>
</tr>
<tr>
<td>3.1.4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>317</td>
<td>6865</td>
<td>8134</td>
<td>8986</td>
<td>31357</td>
<td>35969</td>
<td>3.1722</td>
<td>3.1722</td>
<td>3.1722</td>
<td>5.2322</td>
<td>5.3718</td>
<td>5.5827</td>
</tr>
<tr>
<td>3.1.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6675</td>
<td>7944</td>
<td>8801</td>
<td>31567</td>
<td>35696</td>
<td>3.1722</td>
<td>3.1722</td>
<td>3.1722</td>
<td>5.2322</td>
<td>5.3718</td>
<td>5.5827</td>
</tr>
<tr>
<td>3.2.1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1422</td>
<td>2454</td>
<td>34046</td>
<td>38905</td>
<td>4.2913</td>
<td>3.1545</td>
<td>3.1650</td>
<td>5.5821</td>
<td>5.6649</td>
<td>5.7342</td>
</tr>
<tr>
<td>3.2.2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1109</td>
<td>35422</td>
<td>39805</td>
<td>43774</td>
<td>56637</td>
<td>3.1545</td>
<td>3.1650</td>
<td>5.5821</td>
<td>5.6649</td>
<td>5.7342</td>
</tr>
<tr>
<td>3.2.3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>35326</td>
<td>40456</td>
<td>44213</td>
<td>56926</td>
<td>3.1545</td>
<td>3.1650</td>
<td>5.5821</td>
<td>5.6649</td>
</tr>
<tr>
<td>4.0.1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5783</td>
<td>12233</td>
<td>29680</td>
<td>3.1545</td>
<td>3.1650</td>
<td>5.5821</td>
<td>5.6649</td>
</tr>
<tr>
<td>4.0.2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7026</td>
<td>24900</td>
<td>25927</td>
<td>5.1970</td>
<td>5.1970</td>
<td>5.1970</td>
</tr>
<tr>
<td>4.0.3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>18667</td>
<td>19765</td>
<td>20743</td>
<td>4.7619</td>
<td>4.7619</td>
<td>4.7619</td>
</tr>
<tr>
<td>4.1.1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1172</td>
<td>2472</td>
<td>35067</td>
<td>3.2317</td>
<td>3.2317</td>
</tr>
<tr>
<td>4.1.2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1172</td>
<td>2472</td>
<td>35067</td>
<td>3.2317</td>
<td>3.2317</td>
</tr>
<tr>
<td>4.1.3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>33241</td>
<td>3.2317</td>
<td>3.2317</td>
<td>3.2317</td>
</tr>
<tr>
<td>4.2.1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>33241</td>
<td>3.2317</td>
<td>3.2317</td>
</tr>
</tbody>
</table>

Figure 6.5: Heatmap example
Therefore the distinction between the results of coupling and cohesion metrics is especially important as high cohesion increase of one role indicates high re-work of the modeling tools used by this role while high coupling increase indicates possible interoperability issues between the tools used by different roles.

Apart from the comparison of the metrics’ results calculated on two AUTOSAR meta-model versions, the ARCA tool can be used for generating .csv reports from the comparison of the metrics’ results calculated on multiple meta-model versions specified in the configuration file (i.e., paths to the corresponding .mod files are provided) for all defined roles (see ‘Report metrics’ under 6 in Figure 6.4). Based on these reports, histograms and trend charts showing the evolution of the AUTOSAR meta-model with respect to the analyzed properties can be created for each role.

The main use-case for these charts is during project planning in order to identify places in the development process (i.e., exchange of models between two roles) where additional resources/integration activities should take place. An example of such chart showing the complexity evolution of the AUTOSAR meta-model between releases 3.1.1 - 4.2.1 for 4 roles defined in the configuration file (the roles and their mapping to different packages is described in [6]) based on the results of the Fan-IO metric is shown in Figure 6.6.

![Complexity evolution](image)

Figure 6.6: Complexity evolution

We can see a big increase in complexity between AUTOSAR releases 4.0.1 and 4.1.1 for all roles which indicates higher risk of faults. We can also see that the role of Application software designers is the most complex role.

### 6.4.4 Presenting/quantifying the feature related changes

This functionality is similar to the first functionality, however this time it is possible to present the results of changes for a selected role related to specific feature. This is done utilizing the process presented in Figure 6.2. Based on the provided SVN path to the AUTOSAR meta-model and a list of Bugzilla entries relevant for the analyzed feature (see 4 in Figure 6.4), the tool will analyze the changes between all meta-model commits in which the commit message references at least one Bugzilla entry from the list. In case the message references Bugzilla entries related to more than one feature, the user is asked to discard the changes not relevant for the analyzed feature.
The main use-case for this functionality is the impact assessment of adopting only certain features from new AUTOSAR releases on the modeling tools used in the development. This is possible as AUTOSAR features are usually loosely coupled and the implementation of changes is always done in a backwards compatible way so it is not very likely that changes caused by one feature will affect other existing and/or new features. This functionality is important as it is often the case that only certain features from new AUTOSAR releases are actually required by the development projects so there is no need for adopting an entire new release with thousands of meta-model changes.

Apart from the analysis of changes related to a specific AUTOSAR feature, the ARCA tool can be used for generating .csv reports from the analysis of multiple features specified in the configuration file (i.e., relevant Bugzilla entries for each feature are provided) for all defined roles (see ’Report features’ under 6 in Figure 6.4). Based on these reports, histograms showing the number of changes (or the number of modified, added and removed elements, attributes and packages) needed to be implemented in the AUTOSAR based tools to adopt each feature can be created for each role.

The example of such a histogram containing the number of relevant changes for 14 new features (referred to as concepts according to the AUTOSAR terminology) of the AUTOSAR release 4.2.1 is shown in Figure 6.7.

We can see that feature 7 (SupportForPBECUConfig) requires significantly more changes than all other features combined, i.e., it represents an outlier. Without this tool, the amount of work needed to adopt this feature in the development projects could be underestimated.

Together with the generated .csv reports from the analysis of multiple features, ARCA tool can be used to identify and present the optimal sets of new features (referred to as ”solutions”) to be adopted in the development projects based on the two objectives [19]: (i) to maximize the weighted number of features to be adopted and (ii) to minimize the number of changes to the AUTOSAR meta-model caused by the new features. The weight of each feature can be defined in the configuration file.
The optimal solutions are presented on the Pareto optimality chart which is created automatically by running the generated R script using the R tool [20]. The script also shows which features are included in which solution. An example of the Pareto chart with 14 optimal solutions containing features of equal weight from the AUTOSAR release 4.2.1 is presented in Figure 6.8 and the mapping of features to the solutions is presented in Table 6.1.

Table 6.1: Mapping of features to feature sets

<table>
<thead>
<tr>
<th>Feature set</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>9, 11</td>
</tr>
<tr>
<td>3</td>
<td>9, 11, 14</td>
</tr>
<tr>
<td>4</td>
<td>9, 11, 12, 14</td>
</tr>
<tr>
<td>5</td>
<td>9, 11, 12, 13, 14</td>
</tr>
<tr>
<td>6</td>
<td>5, 9, 11, 12, 13, 14</td>
</tr>
<tr>
<td>7</td>
<td>4, 5, 9, 11, 12, 13, 14</td>
</tr>
<tr>
<td>8</td>
<td>2, 4, 5, 9, 11, 12, 13, 14</td>
</tr>
<tr>
<td>9</td>
<td>2, 4, 5, 8, 9, 11, 12, 13, 14</td>
</tr>
<tr>
<td>10</td>
<td>2, 3, 4, 5, 8, 9, 11, 12, 13, 14</td>
</tr>
<tr>
<td>11</td>
<td>1, 2, 3, 4, 5, 8, 9, 11, 12, 13, 14</td>
</tr>
<tr>
<td>12</td>
<td>1, 2, 3, 4, 5, 6, 8, 9, 11, 12, 13, 14</td>
</tr>
<tr>
<td>13</td>
<td>1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14</td>
</tr>
<tr>
<td>14</td>
<td>1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14</td>
</tr>
</tbody>
</table>

Figure 6.8: Pareto front

We can see that with a relatively small increase in the number of changes, car manufacturers can adopt several additional features (e.g., features 5, 9, 12, 13 and 14 from the solution 6) in comparison to only one feature 11 from the solution 1. However features 6, 10 and especially 7 from the solutions 12, 13 and 14, respectively, require significant re-work in the modeling tools and they should be considered only in case of a high demand.
6.4.5 Combining all tool’s functionalities in car projects

There are still car manufacturers today developing automotive software systems based on an AUTOSAR release 3.x. In order to decide which 4.x release to adopt in the modeling tools used in the development process, the results presented in figures 6.5 and 6.6 can be used. For example if $R_{4.1.1}$ contains the required features, we can see that the increase in the number of changes and complexity between this release and $R_{4.1.3}$ is not that high. Therefore it makes sense to adopt the newer $R_{4.1.3}$ as some of the identified faults in the previous releases have been fixed.

Due to its high increase in the number of changes and complexity, there is a risk of high number of faults in $R_{4.2.1}$ so it may be wise to wait for a future 4.2.x release with stable complexity where most of these faults will be fixed. However if there are some required features from this release, Pareto optimality chart presented in Figure 6.8 can be used to identify which $R_{4.2.1}$ features can be adopted on top of $R_{4.1.3}$ without high risk of late faults and increased cost. These kinds of analysis would not be feasible for the automotive software designers without the tool support that $ARCA$ provides.

6.5 Conclusion

Automotive software designers today need to know AUTOSAR in order to keep track of the change in its meta-model and analyze their impact on different modeling tools used in the development. With the $ARCA$ tool, they can focus on the implementation of car functionalities and get a quick feedback about the cost of switching to a newer AUTOSAR release. Additionally, linking meta-model changes to different features enables a quick overview of the optimal sets of AUTOSAR features to be adopted in the development projects.

As the trend in the size and complexity increase of the AUTOSAR meta-model is expected to continue in future, tools like this become even more important for the car manufacturers in order to assure high quality of the automotive electrical systems and lower their development cost.

The $ARCA$ tool can be downloaded from the following link: http://web.student.chalmers.se/~durisic/ARCA.zip
Bibliography


