UNIVERSITY OF GOTHENBURG

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

Master of Science Thesis in Software Engineering and Management

DARKO DURISIC

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, June 2011

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

DARKO DURISIC

Supervisor: MIROSLAW STARON

Examiner: GERARDO SCHNEIDER

University of Gothenburg

Chalmers University of Technology

Department of Computer Science and Engineering

SE-412 96 Göteborg

Sweden

Telephone + 46 (0)31-772 1000

Cover: Changes are essential in cars’ evolution process.

Department of Computer Science and Engineering

Göteborg, Sweden June 2011.

Acknowledgements

First, I would like to express my deepest gratitude to Dr. Miroslaw Staron, who patiently guided me through the entire research and always inspired to do better.

Then, I would like to thank equally important persons credited for this work - Martin Nilsson, Peter Nordkam and Göran Lundqvist from Volvo, who were always eager to discuss my findings and tirelessly read all my reports providing me with prompt feedback. Additionally, there are many others from Volvo to whom I owe great thanks for the time spent in explaining different things to me and giving me constructive ideas.

Finally, I am infinitely grateful to my family and SS, who were always there for me when I needed them most. I dedicate my entire research described in this thesis to my brother Bogdan.

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

Darko Durisic
Department of Computer Science and Engineering
Chalmers and Gothenburg University
Gothenburg, Sweden

gusdurida@student.gu.se

ABSTRACT

BACKGROUND: In the past few decades, exponential increase in the amount of software used in cars has been recorded. Complex software is hard to maintain, especially due to constant changes which are essential in a car evolution process. To avoid the possible negative impact of changes on the system quality attributes, appropriate measurements of change are needed.

METHOD: The research presented in this paper is based on the quantitative case study conducted together with our industrial partner Volvo Car Corporation (VCC) [1].

RESULTS: The structural complexity and coupling analysis of automotive software systems compared through different releases are applicable for measuring the size and locating the origin of the biggest and the most severe architectural changes.

CONCLUSION: By applying the metrics after each significant change in the system, it is possible to verify that certain quality attributes have not decreased to an unsatisfactory level and to identify parts of the system which should be tested more. This increases the product quality and reduces its development cost.

Keywords

Automotive software, product quality, quality metrics, architectural change, maintainability, complexity, coupling.

1. INTRODUCTION

The amount of software in today's cars has reached one gigabyte of on board binary code (excluding the infotainment domain), and is constantly increasing [2]. Research shows that more than 80% of innovations in cars are related to software and the majority of them are increasing the interaction between previously less dependent parts of the system [3]. At the same time, quality demands for safety, reliability and performance must remain high for the whole car product, including software [4]. Most of the quality attributes are improved with the use of software. A good example of this is "Pedestrian detection" technology, as a part of Volvo's safety system, which is able to prevent more than 50% of pedestrian-involved accidents [5]. However, huge binary code increases the probability of fault propagation in already complex automotive software systems[1], resulting in significantly harder integration testing [4]. Additionally, constant changes in the development process may lead the actual implementation of the system away from its design and architectural decisions making validation of the quality attributes extremely difficult.

[1] Automotive software systems realize up to 2000 software-based functions with more than 10% user functions [3].

Still, software changes are essential in a car evolution process and can take place in any stage of the platform's[2] lifecycle [4]. An example of this has been presented in [4] using a car's headlights: the initial software version controlling this unit in a car was implemented just to turn the lights on and off, the second version was able to adjust manually the beam of light and turn it along the vertical axe, while 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 would be very inefficient to wait for the new platform release to implement these types of changes [4]. On the other hand, a platforms' lifecycle is quite long today. Due to the high and relatively cheap competitors on the market, product quality is vital but not sufficient to sell the expensive product. For this reason, and implied by 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.

Apart from their frequency implying the risk of deteriorated quality, software changes in automotive systems can cause two additional problems:

First, integration and regression testing is very hard since most of the software components are developed by different suppliers. 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 [3]. This way of working increases the quality of delivered components since 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.

Second, most of the changes in automotive software systems are either additions or improvements of the existing functionalities represented with new signals on the electronic busses [4]. As such, the majority of them is affecting the communication between different parts of the system and can be classified as architectural changes [6]. Architectural changes are more likely to cause scattering of functionalities through different sub-systems potentially causing serious malfunctions in others [7].

[2] Platform contains software and hardware infrastructure used in a particular car model(s).

For example, one of the most commonly known faults is a car's "no-start" problem when a driver is, for no specific reason, unable to start the engine of the car, doing it normally just a few seconds later. The explanation for this behavior most probably lies in the start-up process which initiates many different checks and at least one of them fails. The reason for this failure could be the existence of an error in one of the sub-systems which might have nothing to do with 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 the abortion of the car's start-up process. This phenomenon known as the "ripple" effect[3] [8] represents one of the biggest threats to software systems and it is significantly increased with the introduction of architectural changes.

Having in mind the necessity and significance of changes from one side, potential problems they might provoke from the other, and constant demand for low cost, it is very hard to approach the quality issues in a good and systematic way. This is why measuring the size and potential impact of changes on other parts of the system could be the key for assuring robustness, reliability and other quality requirements. It is important to gather this information as soon as possible in the development process in order to reduce the number of late changes and lower the production cost. An example of this has been presented in [4]. It explains that being able to foresee the overload of specific electronic bus and deploying some of the software components to another place in the system before sending requirements to suppliers is much cheaper and efficient than sending a change request later. Additionally, applying the metrics which are able to localize the area that suffered most severe changes indicate parts of the system which should be tested more in order to eliminate potential "ripple effects" [4].

Several metrics 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 most applicable ones to embedded automotive software systems – one based on modules’ complexity and one based on modules’ coupling. We also explain that the measurement results should be compared through different system releases (with focus on the difference between the current and future release) in order to be able to capture the size and potential impact of changes. Finally, we suggest how to interpret their results in order to come to the correct conclusion which should imply the future steps towards securing the desired quality. Since our metrics should be applied in the early stages of the development process (before sending change requests to suppliers) where not many behavioral properties of the system are known, they are mostly focused on structural system properties such as inter-module communication [4]. Still, they can identify early which parts of the system will be affected by changes which can significantly reduce the production cost as well [4].

The rest of the paper is organized as follows: Section 2 describes the related work. Section 3 describes our research method. Section 4 describes the organization of the studied automotive software system at VCC. Section 5 describes the quality metrics applicable to measure the complexity and coupling of automotive software systems. Section 6 describes the suggested way to present measurement results and how they should be interpreted. Section 7 describes the example of the automotive software system and demonstrates the use of presented metrics. Section 8 describes the theoretical and empirical validation of the metrics and Section 9 describes the conclusions and discusses the future work.

2. RELATED WORK

There have been several attempts to measure the size of architectural changes in software systems. One of the most interesting ones is described in [9] where authors try to measure the distance between architectures through different system releases, based on the chosen architectural properties. Also, several researches tried to perform change impact analysis on the architectural level based on the dependencies between architectural units, such as [10] and [11]. However, we are not aware of any attempts to approach change impact analyses from the architectural point of view, based on the complexity and coupling increase in the system through system releases.

There are many different metrics used to measure the complexity and coupling in software systems. Generally, coupling metrics are based on inter-module relations, but complexity metrics can be based on either intra-module relations, inter-module relations (structural complexity), or both [12]. Since this paper observes automotive software systems from the perspective of OEMs[4], it is not possible to apply most of the intra-module complexity metrics available today since they are based on a source code analyses (such as lines of code, the number of operators and operands [13], control graphs [14], syntactic constructs [15], etc. [12]). However, information about the modules and their communication interfaces is available very early (on a design level) and that is why we based our metrics mostly on these structural system properties. An alternative approach to this could be the use of FPA (Function Point Analysis) [16], where each function would be assigned to one or more system modules. Then, the complexity of one module can be calculated as a sum of complexities of all of the functions assigned to it.

The original measure behind our metrics is the strength of module dependencies, as introduced by Stevens et al. [17]. Since then, many different metrics based on this have been introduced such as [18], [19], [20] and [21], especially with the evolution of object-oriented software systems [22]. Some of them rely on the data obtained from source code (such as the number of input-output (IO) variables and methods invoked). Other metrics more interesting for this research focus strictly on the dependences between modules and the information exchange between them - denoted as structural metrics [12]. Probably the most widely accepted structural metric is the one based on modules’ fan-in and fan-out introduced by Henry et al. [23], and it was our major inspiration for defining the complexity model. The coupling model was inspired by the Package Coupling Metrics (PCM) defined by Gupta et al. [24].

Despite the fact that there exist a lot of books and papers related to the complexity and coupling of software systems, we were

[3] In this context, term “ripple effect” is used when a relatively small fault in one part of the system might manifest as a huge malfunction in another.

[4] Majority of modules are developed by suppliers and delivered to OEMs as a "black box" platform specific executable code.

unable to find many of these related to the automotive domain. This kind of specialized approach is important for several reasons such as hierarchical organization of automotive software architecture, distributed development of components, timing constraints in communication between components and prioritization of non-functional requirements where safety and cost have top priority. Most of the things we found related to the automotive domain were related to the AUTOSAR[5] [25] and the principle of complex function decomposition using different software components. We also found many tools available to support the design, implementation and testing of 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.

3. RESEARCH METHOD

According to [26], the formal definition of our research goal is defined as: *Analyze the automotive software system for the purpose of measuring the effect of changes to its architectural properties, with respect to maintainability, robustness, reliability and cost, 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 is conducted using the empirical research method [27] [28] based on the quantitative approach [29]. We first studied the organization of automotive software systems and development process used at VCC [1], with the aim to identify cause-effect relationships between the risk of deteriorated quality and architectural changes. Our hypothesis was based on the assumption that an early measurement of size and impact of changes (before their realization by suppliers) can be helpful in order to avoid potentially bad architectural and design decisions which could affect the product quality and thus reduce the production cost.

After defining the research goal and hypothesis, we conducted a thorough case study analysis [30] and tested the applicability of several different metrics. We concluded, together with our industrial partners from Volvo, that metrics based on the structural complexity and coupling increase in the system are the most suitable ones[6]. In addition, since none of the existing ones were entirely applicable to the automotive domain or did not use the specific characteristics of automotive software systems in order to produce the most correct results, we had to modify the chosen metrics without changing their main logic explained by the authors.

All data used in this study is provided by VCC and is based on the several software platforms deployed to different types of Volvo cars. In order to perform the measurements and present their results, a tool has been implemented which is able to apply the complexity and coupling metrics described in this paper. Apart from metrics' validation purposes, the tool will be used at Volvo regularly (before the realization of changes) in order to increase the efficiency of the software development process, improve the system quality and reduce the production cost.

The theoretical validation of the measures is done according to the complexity and coupling properties defined by Briand et al. [31] (described more in Section 8.1). The empirical validation is done at VCC and it is based on the measurements' results provided by the implemented tool (described more in Section 8.2). Throughout the entire research, many different workshops and interviews with system architects, software designers and component testers were held at VCC. At the beginning, their purpose was to get familiar with the automotive software development process, system organization and the problems arose from constant changes. Later, their purpose was to interpret the measurements' results and validate them.

Apart from the metrics themselves, the focus of this research was placed on the presentation and interpretation of their results (described more in Section 6). This was also done with a great help of our industrial partners from Volvo.

4. DESIGNING SOFTWARE SYSTEMS AT VCC

Changes in the software systems often involve the introduction of new dependency requirements between two components. It is also possible to modify the existing dependency requirements, or remove some in case they are no longer needed. To better understand the need to measure the size and possible impact of these changes to automotive software systems, it is necessary first to understand their common hierarchical organization. This is important because changes in the higher architectural units and possible faults they might cause usually manifest as a more severe malfunctions in the system, harder to be removed. The studied system is developed at VCC and can be observed from two different views - logical view and pre-deployment view.

4.1 Logical View

The logical view represents a hierarchical organization of software components, sub-systems and domains (an example is shown in Figure 1). Software components are the smallest architectural units grouped into sub-systems mostly according to their functionalities and interaction between themselves [4]. In the logical view, they communicate by sending/receiving logical signals. At the top level, the automotive software system is usually divided into different domains clustered according to their application area and associated quality requirements [3]. Each domain contains number of sub-systems and it is possible to have different levels of sub-systems and software components as well. 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 (cruising, auto-braking) and passive (air-bags, belts) safety.
4. Management – contains the common vehicle sub-systems used by all domains such as settings, diagnostics, etc.
5. Human-Machine Interface – contains the sub-systems responsible for interaction between users and the vehicle.
6. Infotainment – contains the information and entertainment sub-systems such as navigation, telephone, etc.

[5] AUTOSAR - AUTomotive Open System Architecture is a standard developed by OEMs, suppliers and tool developers in order to improve the development process and system quality.

[6] One of the main reasons for focusing on the structural metrics is the necessity to apply them early.

Figure 1 shows an example of the logical view of one small part of the system containing one domain (*SafetyControl*), two sub-systems (*PedestrianDetection* and *SafetyHandler*) and 3 software components (*PedestrianDetector*, *PedestrianManager* and *SafetyBrakeManager*). Both *PedestrianDetection* and *SafetyHandler* sub-systems belong to *SafetyControl* domain. *PedestrianDetector* and *PedestrianManager* software components belong to *PedestrianDetection* sub-system, while *SafetyBrakeManager* software component belongs to *SafetyHandler* sub-system. The example is made for the purposes of this paper in order to explain better the common organization of automotive software systems and does not reflect a part of a real system used at VCC.

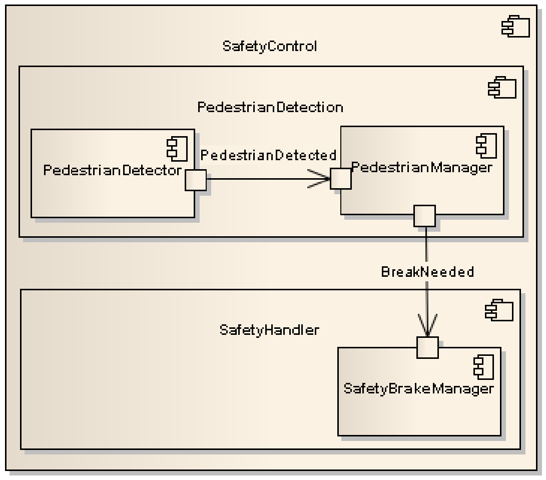

Figure 1: Example of the logical view

Considering the logical view organization of the automotive software system explained in this section, the following can be changed: The addition/removal of a signal between the existing logical software components, the addition/removal of a logical software component with its signals, the addition/removal of a sub-system with its components and the addition/removal of an entire domain[7] with its sub-systems. Additionally, software components/sub-systems can be moved to other sub-systems/domains, respectively.

4.2 Pre-Deployment View

The pre-deployment view has two purposes: First, to show the network topology of ECUs[8] and second, to show the initial deployment of software components to particular ones. Different ECUs are connected via electronic system buses (mostly CAN, LIN, MOST and flex-ray), and they very often work together in order to accomplish one functionality [3]. 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 shows an example of the network topology containing 2 domain ECUs (*SafetyMaster* and *InfotainmentMaster*) and 5 other ECUs (*Radar*, *Camera*, *NightVision*, *TV* and *Radio*). *SafetyMaster* and *InfotainmentMaster*, as domain ECUs, are connected via flex-ray electronic bus. *Radar*, *Camera* and *NightVision* ECUs belong to *SafetyMaster* domain. *Radar* and *Camera* are connected via CAN, while *NightVision* is connected to *Camera* via LIN bus. *TV* and *Radio* ECUs belong to the *InfotainmentMaster* domain, so they are connected via MOST.

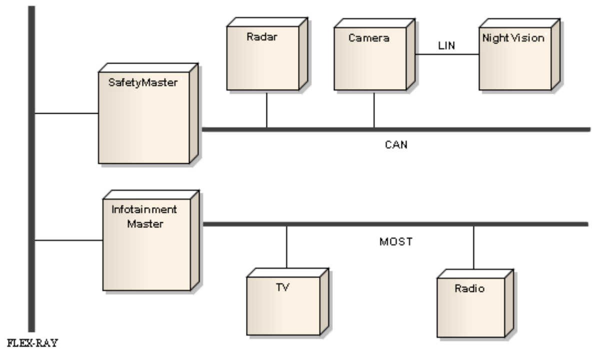

Figure 2: Example of the network topology

Each software component in the logical view is pre-deployed to one ECU. Since suppliers may realize software components differently, the actual deployment can be seen only after they are delivered by suppliers and this is the reason for naming this view – "pre-deployment" view. Often, the decision where one component will be pre-deployed is not made according to their functionalities, but other reasons such as vicinity to hardware (sensors, actuators and buses) or bus load [4]. 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 pre-deployed to different ECUs communicate via ports by sending/receiving system signals.

Figure 3 shows an example of the pre-deployment system view for the logical view shown in Figure 1. Logical software components *PedestrianDetector*, *PedestrianManager* and *SafetyBrakeManager* are mapped to the pre-deployed software components with the same names, and they are all pre-deployed to *SafetyMaster* ECU.

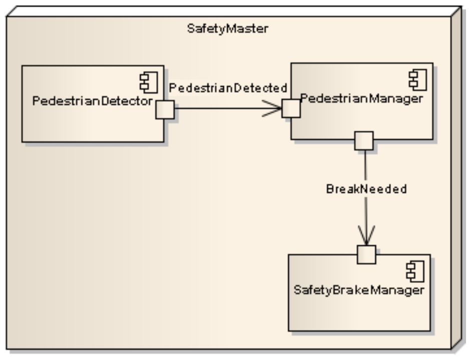

Figure 3: Example of the pre-deployment view

[7] Note that the addition/removal of domains is not very common during the life-span of one platform, but these changes are rather introduced with the release of the new one.

[8] ECU (Electronic Control Unit) represents embedded software system in charge of one or more electrical systems in a platform. Typically inside a car, there exist 70-100 ECUs.

Considering the pre-deployment view organization of the automotive software system explained in this section, the following can be changed: The addition/removal of a signal between the existing pre-deployed software components, the addition/removal of a pre-deployed software component with its signals and the addition/removal of an ECU with its components. Additionally, software components can be moved to other ECUs.

5. QUALITY METRICS

When an architectural change occurs, it could be quite valuable to see how the level of complexity and coupling increases 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 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 4, it is possible to measure structural complexity and coupling of modules based on the strength of their dependences, after each significant change in the system. Moreover, this process can be completely automated in case of unified approach used by OEMs to store dependency requirements.

We define our complexity and coupling measures according to the properties of complexity and coupling measures defined in [31], as explained in Section 8. Generally, the complexity of one component captures the strength of its dependencies towards all 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 components which belong to different modules.

Our quality metrics are based on the increase/decrease in the modules' complexity and coupling through different system releases. Additionally, the comparison between the results of two metrics in the same release is also taken into consideration when defining the future strategies for securing the quality requirements of the system (explained more in Section 6). Since automotive software systems can be observed from two different views - the logical view and the pre-deployment view, both complexity and coupling measures can be applied to both views. This is why we divided our measures into the logical view measures and the pre-deployment view measures. We present in Section 5.1 the logic behind the logical view complexity measure defined in formulas (4) and (5) and coupling measure defined in formula (9). Due to their similarity, we present in Section 5.2 the necessary modifications in order to define the pre-deployment view complexity and coupling measures.

5.1 Logical View Measures

One of the best known structural complexity metrics focused on inter-module complexity is the one defined by Henry et al. based on modules' fan-in and fan-out [23]. 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 defined as:

$$\text{(1a)}\quad C_i = \left(fin_i * fout_i \right)^2 \quad [23]$$

where fin_i represents fan-in of module i, $fout_i$ fan-out of module i and C_i its complexity.

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. Still, 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 other software components in the system (input complexity) and fan-out as the number of transmitted signals to other software components in the system (output complexity). Based on the logic of fin and fout, we can define cin and cout to be input and output complexities of one software component based on one or more complexity attributes (not just the number of sent/received signals) such as hierarchical level of signals, timing constraints, etc [4]. Additionally as explained in [4], we can omit the exponent 2 from formula (1a) due to its unjustified amplification of measurement results. Now, we can calculate single component's complexity in the following way:

$$\text{(1b)}\quad C_i = cin_i * cout_i \quad [4]$$

The overall sub-system, domain and system complexity can be defined as a sum of all components' complexities (C_n) in a sub-system, domain or system, respectively, with n modules:

$$\text{(2)}\quad C_n = \sum_{i=1}^{n} C_i \quad [4]$$

Due to the size of automotive software systems, measuring the overall system complexity increase does not provide very useful results, since a small change in one part of the system will not affect the entire system much [4]. Imagine a change has been made and new signal has been introduced between two software components in the system. If those components previously had relatively low complexity, it might get noticeable higher now after the change is implemented. At the same time in case it was the only change in the system, overall system complexity will not change much. This is why another approach concerning specific inter-component and inter-sub-system dependences must be applied in order to produce valuable results.

One of the solutions is to use Dependency Structure Matrix (DSM) [32] in order to present the relations between different components in the system. DSM was originally created to optimize product development process and show task dependencies, but it can also be applied to software architecture. It preserves components' hierarchy and it is able to show inter-module and intra-module dependencies in a visible way. Additionally, it can be used for other analyses such as identification of architectural patterns. Also, different tools can be found to support these analyses [32].

DSM is a square matrix where each component is assigned to one row and column with the same index. Each $DSM_{i,j}$ field in the matrix has value 1 if there is at least one signal sent from component assigned to row i to component assigned to column j of the DSM, or 0 otherwise. However, this value does not contain any quantitative (such as the number of exchanged signals) and/or qualitative (such as the hierarchical level of the signal) attributes which would more precisely estimate the strength of dependency between them. This is why we suggest the use of Complexity Structure Matrix (CSM) instead [4].

CSM is a square matrix where each software component (as the lowest hierarchical unit) in the system is assigned to one row and column with the same index (like DSM), but its fields contain a value derived from a formula (CSM formula) calculating the strength of dependency between components [4]. The example is presented in [4]: Two software components which are exchanging multiple signals should have higher complexity value than the ones exchanging only one signal. For this reason, $CSM_{i,j}$ should contain a value derived from a formula (from now on referred to as the CSM formula) which calculates the number of sent signals from component assigned to row i to component assigned to column j of the CSM. Then, new dependency on higher hierarchical level (between domains) increases the complexity more than new dependency on lower hierarchical level (between sub-systems), and this should also be included into the CSM formula. Additionally, if other attributes such as signal timing properties (period, maximum travel time, etc.) are available, they can also be included into the formula.

It stems from the previous discussion that the CSM formula can contain multiple attributes (the number of exchanged signals, their hierarchical level, timing properties, etc.). Since not all attributes have the same value range[9], it is necessary to scale them to the desired range of values with lower limit set to one. This is important because in best scenario, they should not affect the complexity calculation but can never decrease it. On the other hand, the number of exchanged signals which is the main and as such mandatory attribute can have value zero, if there is no dependency between two software components. For some attributes without a range (such as type of signals), it is necessary to include the weight factor in the CSM formula (inter-sub-system signals weights 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), so the CSM formula looks as follows:

$$\text{(3a)}\quad CSM_{i,j} = \sum_{k=1, i\neq j}^{num} type(k) \quad [4]$$

where num represents the number of signals sent from software component assigned to row i to software component assigned to column j of the CSM and type(k) its weight factor depending on the signal type (intra-sub-system, inter-sub-system, inter-domain). Based on the logic where higher structural units in the hierarchy should exchange less signals, we concluded, together with Volvo experts, that intra-sub-system signals should have weight factor 1, inter-sub-systems signals weight factor 1.3 and inter-domain signals weight factor 1.8 [4].

After creation of CSM, the rest of the complexity calculations can be done automatically. For example, the sum of all elements in column j ($j \neq i$) represents input complexity of the software component assigned to column j of the CSM, while the sum of all elements in row i ($i \neq j$) represents the output complexity of the software component assigned to row i of the CSM.

$$\text{(4)}\quad cout_i = \sum_{j=1, j\neq i}^{n} CSM_{i,j}\ ,\ cin_j = \sum_{i=1, i\neq j}^{n} CSM_{i,j} \quad [4]$$

[9] For example, the number of exchanged signals is usually 1-10, while the signal period is usually 1-1000 milliseconds.

Incorporating formula (4) into formula (1b), a single software component's complexity (C_x) can be calculated as:

$$\text{(5)}\quad C_x = \sum_{j=1, j\neq x}^{n} CSM_{x,j} * \sum_{i=1, i\neq x}^{n} CSM_{i,x}$$

Incorporating formula (5) into formula (2), total complexity of a sub-system, domain or system containing n software components (C_n) can be calculated as:

$$\text{(6)}\quad C_n = \sum_{x=1}^{n} \left[\sum_{j=1, j\neq x}^{n} CSM_{x,j} * \sum_{i=1, i\neq x}^{n} CSM_{i,x} \right]$$

According to formula (6), the explained complexity model includes the internal dependencies between components inside the same sub-systems and domains when calculating their complexity. However, the measure excluding them could also be useful, especially in prediction of possible fault propagations in the system. For this purpose, Package Coupling Metrics (PCM) named and defined by Gupta et al. can be used to supplement the explained complexity measure [24]. According to them, the following formula can be used to calculate the coupling between two packages based on the number of dependencies between the software components contained inside of them (where one component belongs to one package, and the other component belong to the other package on the same hierarchical level):

$$\text{(7)}\quad Coup\left(P_a^l, P_b^l\right) = \sum_{i=1}^{n} \sum_{j=1, j\neq i}^{m} r\left(e_i^{l+1}, e_j^{l+1}\right) + \\ + \sum_{j=1}^{m} \sum_{i=1, i\neq j}^{n} r\left(e_j^{l+1}, e_i^{l+1}\right) \quad [24]$$

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

Total coupling of a single package in the system containing t packages is calculated as:

$$\text{(8)}\quad PCM\left(P_a^l\right) = \sum_{b=1 \wedge b\neq a}^{t} Coup\left(P_a^l, P_b^l\right) \quad [24]$$

Applied to the automotive software systems logical view, CSM can be used as a source for obtaining strengths of dependencies between software components. In this case, based on formulas (7) and (8), the following formula can be used to calculate package coupling of a single sub-system/domain:

$$\text{(9)}\quad PCM\left(P_a^l\right) = \sum_{b=1 \wedge b\neq a}^{t} \left[\begin{array}{l} \sum_{i=1}^{n} \sum_{j=1}^{m} CSM_{ind\left(P_{a\,i}^l\right), ind\left(P_{b\,j}^l\right)} + \\ + \sum_{j=1}^{m} \sum_{i=1}^{n} CSM_{ind\left(P_{b\,j}^l\right), ind\left(P_{a\,i}^l\right)} \end{array} \right] \quad [4]$$

where P_a^l and P_b^l represent two sub-systems/domains on the hierarchical level l, m and n the number of their components and ind function which returns the CSM index assigned to the component inside particular package.

The results of the complexity measures defined in formulas (5) and (6) and coupling measures defined in formula (9) should be compared and analyzed together, as explained in Section 6. The demonstration of the measurement process can be seen in the example provided in Section 7.

5.2 Pre-Deployment View Measures

When creating the requirements specification for suppliers in order to implement particular functionalities, information from the logical view is not sufficient. Suppliers need to know to which ECUs particular software components will be deployed as well. This is mostly due to the existence of several other external requirements such as hardware requirements (CPU frequency, memory consumption, etc.), necessary for suppliers to be aware of while implementing the components. For this reason, it is also important to estimate the potential impact of changes to the network topology in the pre-deployment view as well. This is done by measuring the complexity and coupling increase in the system in a similar way as it was presented for the logical view in Section 5.1.

In the pre-deployment view, logical software components are mapped to pre-deployed software components and the ECUs can be considered as sub-systems containing the pre-deployed components. It stems from the previous that the pre-deployment system organization is very similar to the logical system organization, so both complexity and coupling measures defined for the logical view can be applied here as well. However, formula (3a) used for calculating the strength of dependencies between software components (the CSM formula) has to be modified for two reasons [4]: first, signal types can no longer be intra-sub-system, inter-sub-system or inter-domain, but intra-ECU and inter-ECU instead. Second, 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 should also be included into the CSM formula. The lower the MaxAge value is, the more complex system we have since it is harder to satisfy all timing requirements.

Based on a network topology where system signals and their timing constraints have strong impact on system performance, we concluded, together with Volvo experts, that intra-ECU signals should have weight factor 1, inter-ECU signals weight factor 1.5 and the weight factor for the MaxAge attribute should vary from [1-1.5), depending on its value [4]. Assuming that it ranges from [1-1000] milliseconds, new CSM formula looks as follows:

$$\text{(3b)}\quad CSM_{i,j} = \sum_{k=1, i\neq j}^{num} \left\langle typel(k) * \left(1{,}5 - \frac{MaxAge(k)}{2000ms}\right)\right\rangle \quad [4]$$

where num represents the number of signals sent from the ECU assigned to row i to the ECU assigned to column j of the CSM, type(k) the weight factor depending on the signal type (intra-ECU, inter-ECU), and $MaxAge_{i,j}(k)$ its maximum allowed time to travel between the ECUs assigned to row i and column j of the CSM. MaxAge for intra-ECU signals is set to 1000 milliseconds by default, so it does not affect the calculation.

The rest of the pre-deployment view complexity and coupling measurements can be done in the same way as explained for the logical view, by applying formulas (5), (6) and (9).

6. PRESENTATION AND INTERPRETATION OF RESULTS

6.1 Presentation of Measurement Results

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 so that conclusions can be made quickly. In order to achieve this, we suggest graphical representation of complexity and coupling increase/decrease in the system through different system releases using histograms.

Despite the fact that the explained complexity and coupling measures produce numerical results, they do not represent a strong base for their interpretation. For example, if one sub-system is exchanging 100 different signals with substantial number of other sub-systems, it indicates its very high complexity and coupling value. However, this does not necessarily have to be a sign of bad architecture because the purpose of this sub-system might be to conduct different signals towards destination sub-systems. This is why the presentation of complexity and coupling change through different releases compared with other modules in the system could be much more useful. 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 through releases, the explained measures should be applied after each architecturally significant change [4].

For presenting the level of complexity and coupling increase/decrease in each hierarchical level for both system views[10], we suggest the use of histograms. Several histograms should be used for this purpose and most of them are demonstrated in the example presented in Section 7:

1. Logical software components' complexity change – presents the change in the complexity of all logical software components in the system between previous and current system releases (Figure 8).
2. Logical sub-systems' complexity change – presents the change in the complexity of all sub-systems in the system between previous and current system releases (Figure 9).
3. Logical sub-systems' coupling change – presents the change in the coupling of all sub-systems in the system between previous and current system releases (Figure 10).
4. Logical domains' complexity change – presents the change in the complexity of all domains in the system between previous and current system releases.
5. Logical domains' coupling change – presents the change in the coupling of all domains in the system between previous and current system releases.

[10]The logical system view has three hierarchical levels: logical software components, sub-systems and domains. The pre-deployment system view has two hierarchical levels: pre-deployed software components and ECUs.

6. Pre-deployment software components' complexity change – presents the change in the complexity of all pre-deployed software components in the system between previous and current system releases (Figure 12).
7. Pre-deployment ECUs' complexity change – presents the change in the complexity of all ECUs in the system between previous and current system releases (Figure 13).
8. Pre-deployment ECUs' coupling change – presents the change in the coupling of all ECUs in the system between previous and current system releases (Figure 14).

Note that it is not possible to measure coupling of logical and pre-deployed software components as the smallest architectural units, according to formula (9). In addition to the explained histograms which present complexity and coupling change between previous and current system releases, we suggest the use of Trend charts (Figures 11 and 15). Their purpose is 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 pre-deployed software component or ECU in the pre-deployment view, through all available system releases including the newest one.

6.2 Interpretation of Measurement Results

As a first step when interpreting measurement results, we suggest finding of sub-systems, as logical units, and ECUs, as physical units, which have suffered significant increase in their complexity and/or coupling. After identifying such sub-systems and ECUs, we can go one level lower and see which software components (logical in case of sub-systems and pre-deployment in case of ECUs) are mostly responsible for this increase.

Apart from identifying sub-systems and ECUs which suffered most severe changes, our focus should be on the comparison between complexity and coupling measurement results. The following example illustrates why this is important: imagine that one sub-system has increased in complexity much more than it has increased in coupling. This indicates that changes introduced a lot of new functions assigned to this sub-system. However, they are localized and as such do not represent a huge threat to other parts of the system (not high risk of fault propagation and "ripple" effects). Still, this sub-system should clearly be tested more after the implementation of changes. On the other hand, if one sub-system has increased/decreased in coupling similarly as it has increased/decrease in complexity, this could indicate possible serious architectural changes that might affect many parts in the system. The reason and origin of these types of 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 (or ECU in the pre-deployment view). 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 complexity and/or coupling of one or more parts of the system indicated by measurement results have increased to an unsatisfactory level, it could affect the quality requirements of the entire system (such as maintainability, reliability and robustness). In that case, there are three possible steps that could be taken in order to minimize this risk:

1. Immediate structural recomposition in parts of the system affected by changes before sending requirements for their realization to suppliers. The purpose of this is to balance the complexity and/or coupling in the system[11].
2. Proceeding with implementation of changes having in mind identified problems for future system releases or introduction of new software platform.
3. Proceeding with implementation of changes having in mind sub-system with high complexity and coupling increase in integration and regression testing phases. This knowledge can also reduce the cost of testing.

In addition to this, information about the most variable parts in the system could be used to point out functionally unstable sub-systems that need special attention while tested and/or potential structural recomposition in future.

7. EXAMPLE

In this section, 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 the example system in Section 7.1, then we demonstrate the measurements and present their results in Section 7.2, and finally we discuss the results in Section 7.3.

7.1 The Example System Description

In this section, we show the example of the automotive software system from both logical and pre-deployment views. It is important to understand that despite the fact that it reflects the logic and organization of a real software system used in cars, it does not represent one (or part of it) and it is created for the purpose of understanding better the presented metrics.

Figure 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 car's "Auto-brake" feature when pedestrian is detected in front of the car.

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

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

[11]For example, this could be done by introducing new software components which can take some of the functionalities [4].

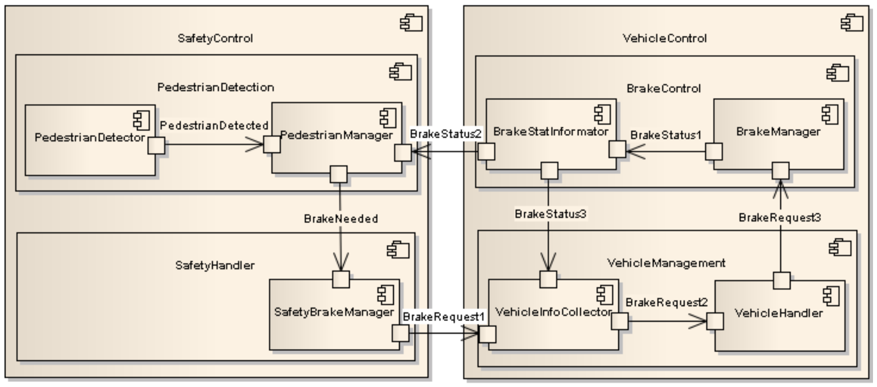

Figure 4: Logical view of the current system release

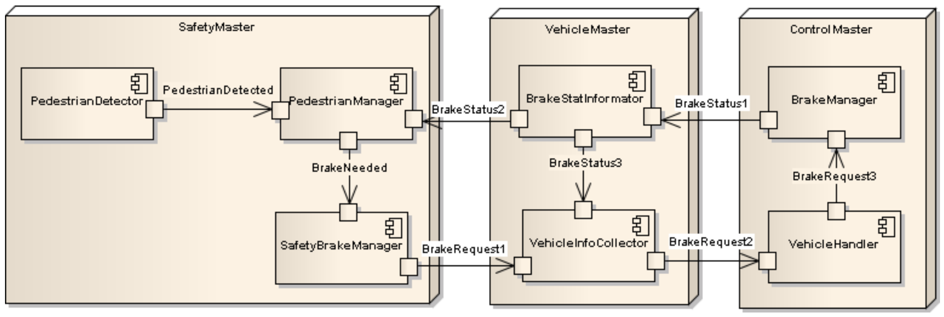

Figure 5: Pre-deployment view of the current system release

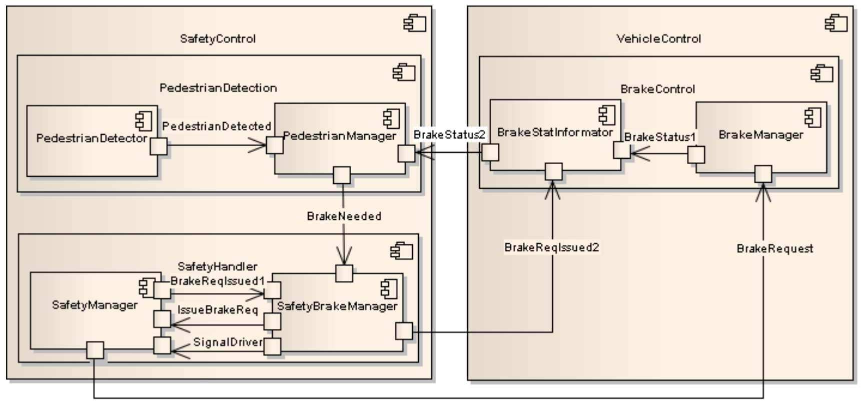

Figure 6: Logical view of the future system release

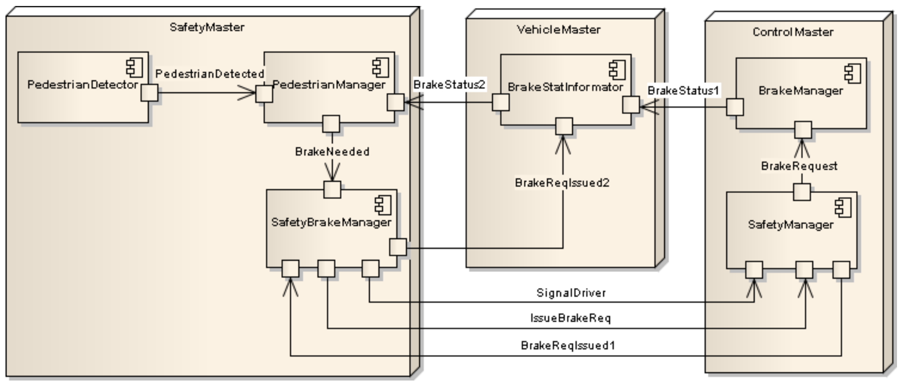

Figure 7: Pre-deployment view of the future system release

Figure 5 shows the pre-deployment view of the current system release before the realization of changes. All software components from the logical view shown in Figure 4 are now pre-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 the two ECUs (MaxAge).

Figure 6 shows the logical view of the future system release after the realization of changes. First, it has been concluded that *VehicleManagement* sub-system inside *VehicleControl* domain is no longer needed and requests for controlling the vehicle should be sent directly to the responsible sub-system (not via *VehicleManagement* as shown in Figure 4). Second, another software component (*SafetyManager*) has been introduced to *SafetyHandler* sub-system which is responsible for warning the passengers and other road users about the safety issues using sound, lights etc.

Figure 7 shows the pre-deployment view of the future system release after the realization of changes. Software components from *VehicleManagement* sub-system are removed from *VehicleMaster* and *ControlMaster* ECUs, and new *SafetyManager* software component added to *SafetyHandler* sub-system is pre-deployed to *SafetyMaster* ECU. The number assigned to each inter-ECU signal represents its MaxAge.

7.2 Measurements and Results Presentation

In this section, we demonstrate the use and presentation of the results of the logical and pre-deployment view complexity and coupling metrics based on the example presented in Section 5.1.

7.2.1 Logical View

Before calculating the complexity and coupling of the logical view software components, sub-systems and domains, it is necessary to create CSM. Since we are applying the complexity and coupling measures in order to present their difference between two releases, two CSMs should be created: one for the system release before the realization of changes, and one for the system release after the realization of changes.

CSM presented in Table 2 corresponds to the current system release shown in Figure 4, and each field in the CSM is calculated using formula (3a) and signal weights defined in Section 5.1. As explained in Section 5.1, each row and column in the matrix with the same index is assigned to one logical software component from the logical view. In this case, CSM indexes assigned to software components from Figure 4 are shown in Table 1 (*PedestrianDetector* software component is assigned to CSM row and column 1, *PedestrianManager* to CSM row and column 2, etc.).

Table 1: CSM assignment of the current release SW components

PedestrianDetector	1
PedestrianManager	2
SafetyBrakeManager	3
BrakeStatInformator	4
BrakeManager	5
VehicleInfoCollector	6
VehicleHandler	7

Table 2: CSM for the logical view current system release

	1	2	3	4	5	6	7
1		1	0	0	0	0	0
2	0		1,3	0	0	0	0
3	0	0		0	0	1,8	0
4	0	1,8	0		0	1,3	0
5	0	0	0	1		0	0
6	0	0	0	0	0		1
7	0	0	0	0	1,3	0	

For example, software component *PedestrianDetector* with CSM index 1 is sending one intra-sub-system signal *PedestrianDetected* to software component *PedestrianManager* (Figure 4) with CSM index 2 (Table 1). According to formula (3a), this implies that $CSM_{1,2}$ field in the matrix should contain value 1, as shown in Table 2. The rest of the CSM fields shown in Table 2 are calculated in the same way.

CSM shown in Table 4 corresponds to the future system release shown in Figure 6. The values of its fields are calculated in the same way as the values of the CSM fields shown in Table 2 for the current system release. CSM indexes assigned to the future release software components are presented in Table 3[12].

Table 3: CSM assignment of the future release SW components

PedestrianDetector	1
PedestrianManager	2
SafetyManager	3
SafetyBrakeManager	4
BrakeStatInformator	5
BrakeManager	6

Table 4: CSM for the logical view future system release

	1	2	3	4	5	6
1		1	0	0	0	0
2	0		0	1,3	0	0
3	0	0		1	0	1,8
4	0	0	2		1,8	0
5	0	1,8	0	0		0
6	0	0	0	0	1	

Applying formulas (5) and (6) described in Section 5.1 to CSMs shown in Tables 2 and 4, it is possible to calculate complexity of all logical software components, sub-systems and domains in both current and future system releases.

Figure 8 shows the complexity difference between the current (in the charts referred to as Release 1) and future (in the charts referred to as Release 2) system releases for all logical software components. Components shown in the horizontal axis in the histogram are ordered by complexity difference between the two releases, where the ones with the highest difference are placed at the beginning in order to be easily noticed.

[12]Note that the numbers assigned for the same logical software components in two releases differ between Tables 1 and 3.

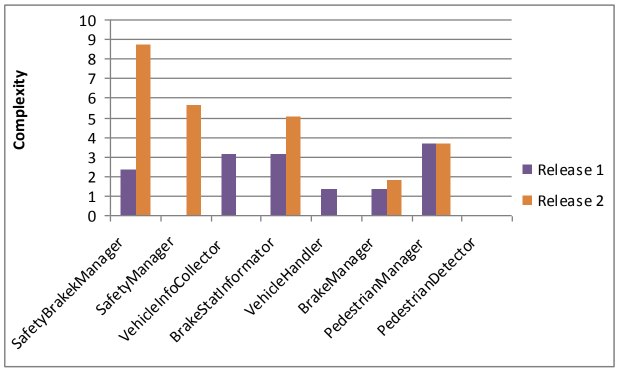

Figure 8: Logical software components' complexity change

For example, according to formula (5) explained in Section 5.1, total complexity of *BrakeManager* software component with CSM index 5 in the current system release (Table 1) is equal to the multiplication of sums of all values in row 5 and column 5 of the CSM shown in Table 2. Therefore, its total complexity equals (0 + 0 + 0 + 1 + 0 + 0) * (0 + 0 + 0 + 0 + 0 + 1.3) = 1.3, as shown in Figure 8. However, complexity of *BrakeManager* software component with CSM index 6 in the future system release (Table 3) is equal to the multiplication of sums of all values in row 6 and column 6 in the CSM shown in Table 4. Therefore, its total complexity equals 1 * 1.8 = 1.8, as shown in Figure 8. Finally, a visible increase of 1.8 - 1.3 = 0.5 of *BrakeManager* software component is also presented in the same figure. The rest of the calculations shown in Figure 8 are done in the same way.

Figure 9 shows the complexity difference between the two releases for all sub-systems. The complexity change of each sub-system is calculated as a sum of complexities of all of its logical software components. For example, sub-system *BrakeControl* in the current system release contains two software components: *BrakeManager* and *BrakeStatInformator*, so its total complexity equals 3.1 + 1.3 = 4.4, as shown in Figure 9. The same logic can be applied to *BrakeControl* sub-system in the future system release, where its total complexity equals 5.04 + 1.8 = 6.84, as shown in Figure 9. Finally, a visible complexity increase of 6.84 - 4.4 = 2.44 of *BrakeControl* sub-system is also presented in the same figure. The rest of the calculations shown in Figure 9 are done in the same way. Sub-systems shown in the horizontal axis in the histogram are ordered by complexity difference.

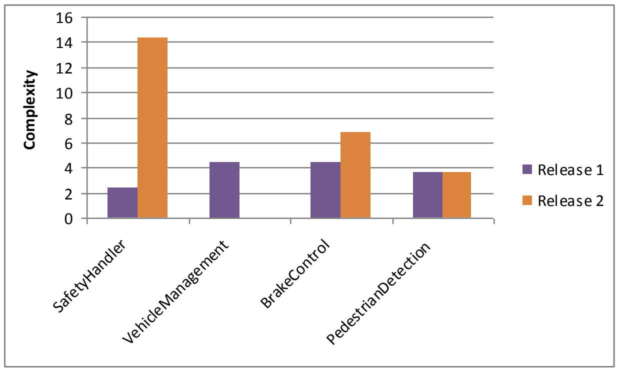

Figure 9: Logical sub-systems' complexity change

Calculating the complexity difference between the current and future system releases for all logical domains can be done in the same way as for the sub-systems. The complexity change of each domain is calculated as a sum of complexities of all of its components and should be presented in a histogram similar to the one shown in Figure 9.

In the logical view, coupling measurements can only be applied to sub-systems and domains (according to formula (9) explained in Section 5.1 which requires a package of components). Figure 10 shows the coupling change for all sub-systems between the current and future system releases, based on CSMs shown in Tables 2 and 4. Sub-systems shown in the horizontal axis in the histogram are ordered by complexity difference.

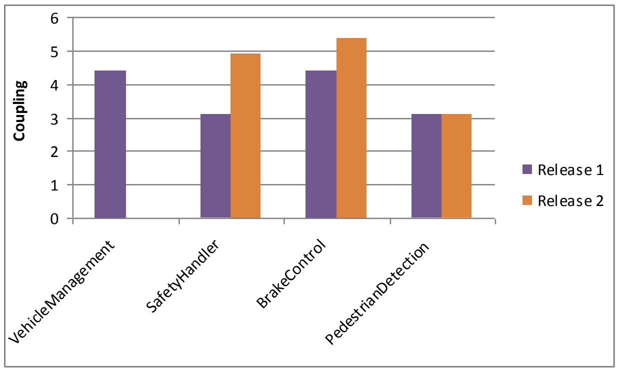

Figure 10: Logical sub-systems' coupling change

For example, sub-system *BrakeControl* in the current system release contains two software components: *BrakeManager* with CSM index 5 and *BrakeStatInformator* with CSM index 4 (Table 1). Its total coupling is equal to the sum of all strengths of dependences between these two components and other components in the system, not counting the strength of dependency between them. In CSM shown in Table 2, the strength of dependency between components is shown in fields $CSM_{4,2}$ (*BrakeStatInformator* → *PedestrianManager*), $CSM_{4,6}$ (*BrakeStatInformator* → *VehicleInfoControl*) and $CSM_{7,5}$ (*VehicleHandler* → *BrakeManager*). Therefore, its total coupling equals 1.8 + 1.3 + 1.3 = 4.4, as shown in Figure 10. After applying the same calculation for *BrakeControl* sub-system in the future system release, its total coupling equals 1.8 + 1.8 + 1.8 = 5.4, as shown in Figure 10. Finally, a visible complexity increase of 5.4 - 4.4 = 1 of *BrakeControl* is also presented in the same figure. The rest of the calculations shown in Figure 10 are done in the same way.

Similarly to this, we can present the coupling change of domains between the current and future system releases with histograms.

The charts shown in Figures 8, 9 and 10 are used to present the complexity or coupling change of all software components, sub-systems and domains between the current and future system releases. However, it could also be useful 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. An example is shown in Figure 11 for the complexity trend of sub-system *BrakeControl* in two releases (in reality, there should be more).

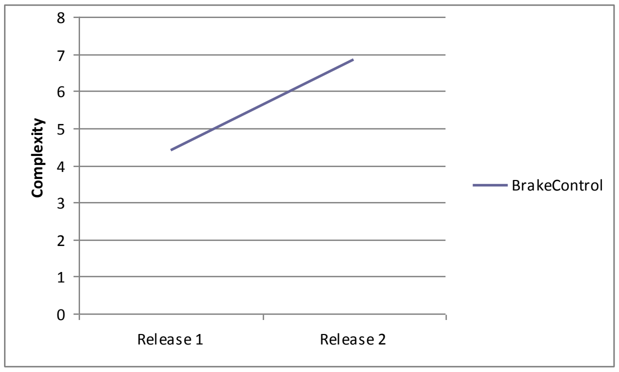

Figure 11: *BrakeControl* sub-system's complexity trend

7.2.2 *Pre-deployment View*

Similarly to the logical view complexity and coupling change measurements, first step in measuring the pre-deployment complexity and coupling change between the two releases is to create two CSMs, one for the current system release and one for the future system release. Since each pre-deployed software component is mapped to a logical software component with the same name, we can use the CSM indexes assigned to logical software components for the pre-deployed software components as well. Therefore, based on the CSM indexes assigned to software components in Table 1, Table 5 shows CSM for the current system release in the pre-deployment view. Based on the CSM indexes assigned to software components in Table 3, Table 6 shows CSM for the future system release in the pre-deployment view. Each field in the two CSMs is calculated using formula (3b) described in Section 5.2.

Table 5: CSM for the pre-deploy. view current system release

	1	2	3	4	5	6	7
1		1	0	0	0	0	0
2	0		1	0	0	0	0
3	0	0		0	0	1,9	0
4	0	1,9	0		0	1	0
5	0	0	0	2,1		0	0
6	0	0	0	0	0		2,1
7	0	0	0	0	1	0	

For example, software component *SafetyBrakeManager* with CSM index 3 is sending one inter-ECU signal *BrakeRequest1* with MaxAge 500 milliseconds to software component *VehicleInfoCollector* (Figure 5) with CSM index 6 (Table 1). According to formula (3b), this implies that $CSM_{3,6}$ field in the matrix should contain value 1.5 * (1.5 – 500 / 2000) $\approx$1.9, as shown in Table 5. The rest of the CSM fields shown in Table 5 are calculated in the same way.

The values of the CSM fields shown in Table 6 for the future system release in the pre-deployment view are calculated in the same way as the values of the CSM fields for the current system release shown in Table 5. Applying formulas (5) and (6) described in Section 5.1 to created CSMs shown in Tables 5 and 6, it is possible to calculate complexity for all pre-deployed software components and ECUs in both current and future system releases.

Table 6: CSM for the pre-deploy. view future system release

	1	2	3	4	5	6
1		1	0	0	0	0
2	0		0	1	0	0
3	0	0		2,1	0	1
4	0	0	4,1		1,9	0
5	0	1,9	0	0		0
6	0	0	0	0	1,5	

Figure 12 shows the complexity difference between the two releases of all pre-deployed software components. Components shown in the horizontal axis in the histogram are ordered by complexity difference between the two releases, where the ones with the highest difference are placed at the beginning.

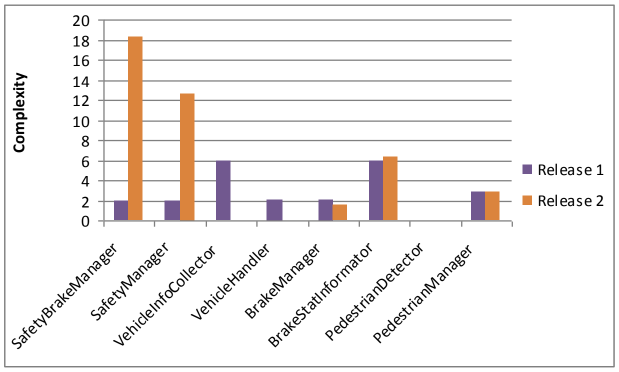

Figure 12: Pre-deployment SW components' complexity change

For example, according to formula (5) explained in Section 5.1, total complexity of *BrakeManager* software component with CSM index 5 in the current system release (Table 1) is equal to the multiplication of sums of all values in row 5 and column 5 of the CSM shown in Table 5. Therefore, its total complexity equals (0 + 0 + 0 + 2.1 + 0 + 0) * (0 + 0 + 0 + 0 + 0 + 1) = 2.1, as shown in Figure 12. However, complexity of *BrakeManager* software component with CSM index 6 in the future system release (Table 3) is equal to the multiplication of sums of all values in row 6 and column 6 in the CSM shown in Table 6. Therefore, its total complexity equals 1 * 1.5 = 1.5, as shown in Figure 12. Finally, a visible increase of 2.1 - 1.5 = 0.6 of *BrakeManager* software component is also presented in the same figure. The rest of the complexity calculations between the releases shown in Figure 12 are done in the same way.

Figure 13 shows the complexity difference between the two releases for all ECUs. The complexity change of each ECU is calculated as a sum of complexities of all of its pre-deployed software components. For example, *ControlMaster* ECU in the current system release contains two software components: *BrakeManager* and *VehicleHandler*, so its total complexity equals 2.1 + 2.1 = 4.2, as shown in Figure 13. The same logic can be applied to *ControlMaster* ECU in the future system release, where its total complexity equals 12.6 + 1.5 = 14.1, as shown in Figure 13. Finally, a visible complexity increase of 14.1 - 4.2 = 9.9 of *ControlMaster* ECU is also presented in the same figure. The rest of the calculations shown in Figure 13 are done in the same way. ECUs shown in the horizontal axis in the histogram are ordered by complexity difference.

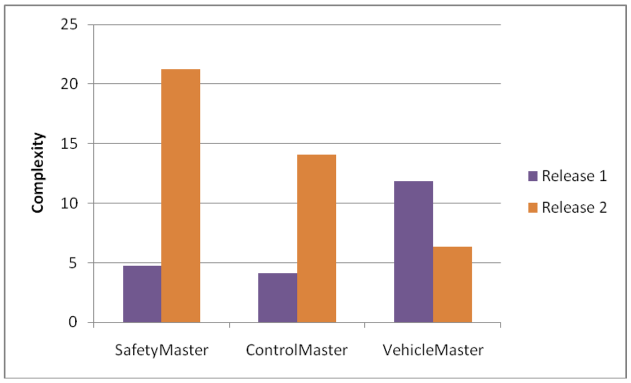

Figure 13: Pre-deployment ECUs' complexity change

In the pre-deployment, coupling measurements can only be applied to ECUs (according to formula (9) explained in Section 5.1 which requires a package of components). Figure 14 shows the coupling change for all ECUs between the current and future system releases, based on CSMs shown in Tables 5 and 6. ECUs shown in the horizontal axis in the histogram are ordered by complexity difference.

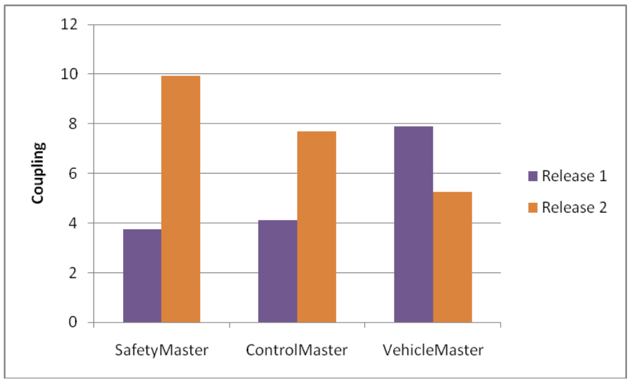

Figure 14: Pre-deployment ECUs' coupling change

For example, *ControlMaster* ECU in the current system release contains two software components: *BrakeManager* with CSM index 5 and *VehicleHandler* with CSM index 7 (Table 1). Its total coupling is equal to the sum of all strengths of dependences between these two components and other components in the system, not counting the strength of dependency between them. In CSM shown in Table 4, the strength of dependency between components is shown in fields $CSM_{5,4}$ (*BrakeManager* → *BrakeStatInformator*) and $CSM_{6,7}$ (*VehicleInfoCollector* → *VehicleHandler*). Therefore, its total coupling equals 2.1 + 2.1 = 4.2, as shown in Figure 14. After applying the same logic for *ControlMaster* ECU in the future system release, its total coupling equals 4.1 + 2.1 + 1.5 = 7.7, as shown in Figure 14. Finally, a visible complexity increase of 7.7 - 4.2 = 3.5 of *ControlMaster* ECU is also shown in the same figure. The rest of the calculations shown in Figure 14 are done similarly.

Following the same logic as for the logical view, it is useful to see the complexity and coupling change of a specific software component or ECU through all system releases in the pre-deployment view as well, using Trend charts. An example is shown in Figure 15 for the coupling trend of ECU *ControlMaster* in two releases (in reality, there should be more).

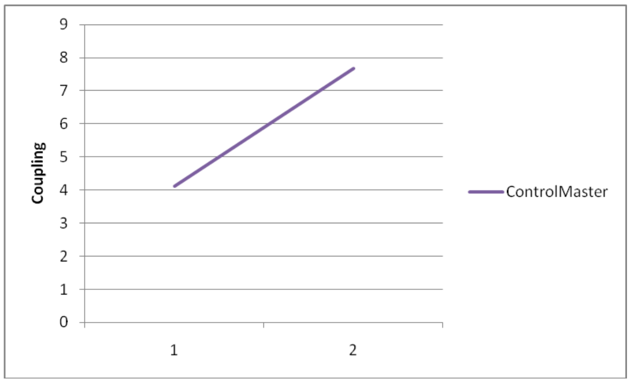

Figure 15: *ControlMaster* ECU's coupling trend

7.3 Results Interpretation

We start the analysis from the logical view. Figure 9 shows much higher increase in the complexity of sub-system *SafetyHandler* in comparison to its coupling increase shown in Figure 10. After looking at the logical software components' complexity change presented in Figure 8, it can be concluded that there are two main reasons for such a high complexity increase of *SafetyHandler*: the addition of new software component *SafetyManager* and the increase in complexity of software component *SafetyBrakeManager*[13]. From the logical point of view, these changes are not truly architectural and can be considered as upgrades of the existing system. This means that eventual faults created inside this sub-system are not very likely to affect the other parts of the system and overall system robustness, concerning the possibility of fault propagations. However, *SafetyHandler* sub-system should be thoroughly tested after the integration and possibly broken into smaller sub-systems in future releases in order to reduce its complexity.

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

After determining the cause for substantial complexity increase of sub-system *SafetyHandler* in logical view and identifying and approving the removal of *VehicleManagement* sub-system, we should look at the effects of these changes to the ECUs in the pre-deployment view. Figure 13 and Figure 14 show the complexity and coupling change of all ECUs in the system. High complexity and coupling increase in *SafetyMaster* ECU is expected, due to new *SafetyManager* software component

[13]This is the consequence of new functionalities added so that drivers and other road users can receive sound and visual information when the safety system is activated (auto-brake).

assigned to it. The same stands for *ControlMaster* ECU, just this time due to the removal of *VehicleManagement* sub-system which increases the number of requests received by software components pre-deployed to *ControlMaster* ECU (signals can now be received from anywhere, not just from the software components inside *VehicleManagement* sub-system). As a consequence of this, the complexity and coupling of *VehicleMaster* ECU is decreased since not so many signals are sent to *VehicleManagement* sub-system anymore. In order to approve these changes for realization, it is necessary to verify that *SafetyMaster* and *ControlMaster* ECUs can handle the new functionalities and signals on the buses from the hardware's perspective (CPU, memory, buses etc.).

As you can see from this brief demonstration of the measurements' results interpretation, the knowledge about the system and experience of the interpreters are very important for making correct conclusions. Still, the real causes for all changes which are making a substantial increase in complexity and coupling of particular sub-system/ECU should be investigated in order to secure the system quality.

8. VALIDATION OF THE METRICS

8.1 Theoretical Validation

In Section 2, two different metrics were proposed: the complexity metric and the coupling metric. In this section, we provide their theoretical validation according to the complexity and coupling properties defined by Briand et al. [31].

The complexity metric holds all five properties of a complexity metric defined in [31]:

1. Non-negativity: The complexity of a system is not negative - the results of both formulas (3a) and (3b) for calculating the CSM fields are non-negative values. In formulas (5) and (6) for calculating complexity, these values are first summed and then 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 of no signals exchanged between modules in the system, corresponding values of the CSM fields are 0, according to formulas (3a) and (3b), resulting in a zero complexity value after applying formulas (5) and (6).
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 a transposed CSM, where the sum of all values in column j corresponds to the sum of all values in row i of the CSM, and vice versa. This does not affect the multiplication in formula (5) for calculating 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 (6), 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" (4).

The coupling metric holds all five properties of a coupling metric defined by Briand et al. [8]:

1. Non-negativity: The coupling of a system is not negative - the results of both formulas (3a) and (3b) for calculating the CSM fields are non-negative values. In formula (9) for calculating coupling, these values are 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 of no signals exchanged between modules in the system, the corresponding values of the CSM fields are 0, according to formulas (3a) and (3b), resulting in a zero coupling value after applying formula (9).
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 (9), 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 (9), validating 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" (4).

8.2 Empirical Validation

Throughout the entire research, regular meetings were held at VCC on a weekly basis in order to discuss our findings, where system architects, designers and testers from Volvo actively participated. All conclusions were validated with them.

In order to validate the results of our metrics based on the complexity and coupling increase in the system through different releases, a software tool has been implemented. The tool is able to extract from the VCC internal database the structural data about the logical software components and sub-systems in the logical view, and the pre-deployed software components and ECUs in pre-deployment view, for the chosen platform. The data is stored internally in order to easily apply the complexity and coupling metrics later[14]. The tool is also able to present the measurement results, as explained in Section 6.

After extracting the data from several software platforms and applying the metrics based on two different releases, the results were presented to the experts in the area of software architecture, design and testing from VCC. First, it was concluded that the complexity and coupling metrics can be applied early in the development process before the realization of changes, and that they are able to identify the most complex parts of the system. Second, it was concluded that the metrics are able to measure the size and locate the origin of the most severe architectural changes in the system and present the

[14]Note that logical domains are not considered in the tool, but they can be approached in the same way as sub-systems.

results in an understandable way. Finally, it was concluded that the suggested interpretation of measurement results can lead to conclusions which can be used to reduce the risk of deteriorated quality, and reduce the development cost by identifying parts of the system which should be tested more.

The tool will continue to be used as a part of the verification process at VCC in order to verify quality strategies related to the complexity of automotive software systems.

9. CONCLUSIONS

In this paper, we tried to emphasize the importance of the change management process in the development of automotive software systems, especially regarding the architecturally significant changes. In order to improve the quality of the system (maintainability, robustness, reliability, etc.), we suggested the use of two quality metrics which are able to measure the size and impact of changes to the complexity and coupling properties of the system. Both metrics are based on already existing and theoretically and empirically validated complexity and coupling measures defined in [23] and [24], respectively, but modified in order to suite hierarchical organization of automotive software systems (seen from two different views –logical view and pre-deployment view) as shown in Section 5. Also, they are designed to support early stages of the development process 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 their results, as equally important segments in the decision making process. We suggested graphical representation of measurement results based on the increase/decrease in the complexity and coupling through different system releases. We also argued that apart from looking at the results separately for each hierarchical level, they should be compared between different levels in the same system release. Based on the measurements' results, future steps towards securing the quality requirements of the system might involve architectural recomposition inside the system or more thorough testing of the parts affected by changes. Finally, we stressed that despite the entirely automated process of measurements and results presentation, human knowledge about the system and experience play a major role in their interpretation. Common automotive system organization, complexity and coupling measures, measurement process and their results presentation and interpretation are all demonstrated in Section 7, based on the example specially designed for the purpose of this paper.

The presented metrics are theoretically validated according to the complexity and coupling properties defined in [31]. Both metrics and the significance of their results have been empirically validated on the software systems used at Volvo cars with the help of software architects, designers and testers from Volvo Car Corporation [1]. However, it is possible that the metrics are applicable to a wider range of software systems which rely on communication between different modules over multiplex buses.

There is still a lot of space for future research in the area of architectural changes in the automotive software industry. For example, it could be valuable to measure the architectural distance between releases as defined in [9], based on the structural properties presented in this paper (the number of signals exchanged, signal type, signal MaxAge, etc.). Then, it would be interesting to test the applicability of the hierarchical analysis presented in [32] on Complexity Structure Matrix (CSM) used in this paper. Finally, including the behavioral aspects of the automotive software system[15] into the account opens a whole new area for more profound change impact analyses.

Since our metrics are based on the structural system properties, it would be interesting to compare their results with the results of Function Point Analysis (FPA) [16], because both metrics can be applied early in the development process based on the system requirements. Additionally, apart from measuring the complexity and coupling of components as explained in this paper, it could also be useful to measure their cohesion, based on the dependencies between software components inside the same module. These results could be used in order to make an additional validation of the conclusions made after the interpretation of the complexity and coupling metrics.

REFERENCES

[1] "Volvo Car Corporation", www.volvocars.com, 2011

[2] A. Pretschner, M. Broy, I. H. Kruger, T. Stauner, "Software Engineering for Automotive Systems: A Roadmap", *Proceedings of the FOSE'07 Conference on Future of Software Engineering*, pp. 55-71, 2007

[3] M. Broy, I. H. Kruger, A. Pretschner and C. Salzmann, "Engineering Automotive Software", *Proceedings of the Conference on IEEE*, vol. 95(2), pp. 356-373, 2007

[4] D. Durisic, M. Staron, M. Nilsson, "Measuring the Size of Changes in Automotive Software Systems and their Impact on Product Quality", *Proceedings of the PROFES'11 12th International Conference on Product Software Development and Process Improvement*, vol. 2, 2011

[5] "Volvo Car Safety Technology - Pedestrian Detection", https://www.media.volvocars.com/us/enhanced/en-us/Media/Preview.aspx?mediaid=31773, 2010

[6] B. J. Williams, J. C. Carver, "Characterizing Software Architecture Changes: A Systematic Review", *Journal of Information and Software Technology*, vol. 52(1), pp. 31-51, 2010

[7] L. Li, A. J. Offutt, "Algorithmic Analysis of the Impact of Changes to Object-Oriented Software", *Proceedings of the ICSM'96 International Conference on Software Maintenance*, pp. 171-184, 1996

[8] L. C. Briand, J. Wuest, H. Lounis, "Using Coupling Measurement for Impact Analysis in Object-Oriented Systems", *Proceedings of the ICSM'99 International Conference on Software Maintenance*, pp. 475, 1999

[9] T. Nakamura, V. R. Basili, "Metrics of Software Architecture Changes Based on Structural Distance", *Proceedings of the METRICS'05 11th IEEE International Software Metrics Symposium*, pp. 8, 2005

[10] J. Zhao, H. Yang, L. Xiang, B. Xu, "Change Impact Analysis to Support Architectural Evolution", *Journal of*

[15]For example, which parts of the system work together in order to fulfill one system functionality?

Software Maintenance: Research and Practice - Special Issue: Separation of Concerns for Software Evolution, vol. 14(5), pp. 317-333, 2002

[11] J. A. Stafford, D. J. Richardson, A. L. Wolf, "Aladdin: A Tool for Architecture-Level Dependence Analysis of Software Systems", *Department of Computer Science, University of Colorado, Boulder*, 1998.

[12] S. H. Kan, "Metrics and Models in Software Quality Engineering", 2nd edition, *Addison Wesley*, 2002

[13] M. H. Halstead, "Elements of Software Science (Operating and Programming Systems Series)", *Elsevier Science*, 1977

[14] T. J. McCabe, "A Complexity Measure", *Proceedings of the ICSE'76 2nd International Conference on Software Engineering*, pp. 407, 1976

[15] L. H. Binder, J. H. Poore, " Field experiments with local software quality metrics ", *Journal of Software: Practice & Experience*, vol. 22 (7), pp. 631-647, 1990

[16] C. Jones, "Estimating Software Costs: Bringing Realism to Estimating", 2nd edition, *McGraw-Hill*, 2007

[17] V. P. Stevens, G. J. Myers, L. L. Constantine, "Structured Design", *IBM Systems Journal*, vol. 13(2), pp. 115-139, 1974

[18] C. L. McClure, "A Model for Program Complexity Analysis", *Proceedings of the ICSE'78 3rd International Conference on Software Engineering*, pp. 149-157, 1978

[19] L.A. Belady, C.J. Evangelisti, "System Partitioning and its Measure", *Journal of Systems and Software*, vol. 2, pp. 23-29, 1981

[20] S. S. Yau, J. S. Collofello, "Some Stability Measures for Software Maintenance", *Journal of IEEE Transactions on Software Engineering*, vol. 6(6), pp. 545-552, 1990

[21] D. N. Card, R. L. Glass, " Measuring Software Design Quality ", *Prentice-Hall*, 1990

[22] W. Li, S. Henry, "Object Oriented Metrics Which Predict Maintainability", *Journal of Systems and Software - Special Issue on Object-Oriented Software*, vol. 23(2), pp 111-122, 1993

[23] S. Henry, D. Kafura, "Software Structure Metrics Based on Information Flow", *Journal of IEEE Transactions on Software Engineering*, vol. 7(5), pp. 510-518, 1981

[24] V. Gupta, J. K. Chhabra, "Package Coupling Measurement in Object-Oriented Software", *Journal of Computer Science and Technology*, vol. 24(2), pp. 273-283, 2009

[25] "AUTOSAR", www.autosar.org, 2011

[26] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslen, "Experimentation in Software Engineering: An Introduction", *Kluwer Academic Publishers*, 2000

[27] R. Glass, "The Software Research Crisis", *Journal of IEEE Software*, vol. 11(6), pp. 42-47, 1994

[28] B. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. E. Emam, J. Rosenberg, "Preliminary Guidelines for Empirical Research in Software Engineering", *Journal of IEEE Transactions on Software Engineering*, vol. 28 (8), pp. 721-734, 2002

[29] J. W. Creswell, "Research Design", "Qualitative and Quantitative Approaches", *Sage*, 1994

[30] P. Runeson, M. Höst, "Guidelines for Conducting and Reporting Case Study Research in Software Engineering", *Journal of Empirical software Engineering*, vol. 14 (2), pp. 137-164, 2009

[31] L. C. Briand, S. Morasca, V. R. Basili, "Property-Based Software Engineering Measurement", *Journal of IEEE Transactions on Software Engineering*, vol. 22(1), pp. 68-86, 1996

[32] N. Sangal, E. Jordan, V. Sinha, D. Jackson, "Using Dependency Models to Manage Complex Software Architecture", Proceedings of OOPSLA'05 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, vol. 40(10), pp. 167-176, 2005