Institute of Software Technology
Reliable Software Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Integration Model for automated Model Generation from Source Code based on AUTOSAR

Shoma Kaiser

Course of Study:	Informatik
Examiner:	Prof. Dr.-Ing. Steffen Becker
Supervisor:	Prof. Dr.-Ing. Steffen Becker Emre Balli, M.Sc. (Daimler AG)
Commenced:	Juli 4, 2019
Completed:	Januar 6, 2020
CR-Classification:	I.7.2

Abstract

Nowadays software development in automotive industry plays an important role. Increasing networking between vehicle systems and large number of software components lead to high complex systems. A modern high-end car has up to 100 Million lines of code [GTD15]. To master this complexity in the automotive industry, model-driven software development has been introduced. A model is an abstraction of the software. It can be applied on different abstraction levels from different viewpoints. Models are necessary to simplify and develop software.

Besides that, in the automotive industry AUTOSAR has been introduced too. The main goal of AUTOSAR is to improve the complexity management of vehicles integrated E/E architectures through increased reuse and exchangeability of software modules.
In reality the model-driven software development is not continuously introduced yet. The software development process is just used partly so that components of the development process are incomplete and inconsistent. This leads to the risk of information loss, errors, missing transparency and overall to lose against the complexity. That is because software changes are implemented in source code but not into the corresponding models. Thus, a part of software diagrams in the model-driven software development process are not consistent and model-driven software development not realized continuously.

This thesis aims to develop an integration model which applies reverse engineering with modifications to generate software models based on source code. In this integration model AUTOSAR should be considered too. Main goal of the integration model is to automatically generate software architecture models. Goal of these models is to increase source code readability, improve software quality and yield advantages for software engineers. The models should guarantee no information loss, correctness and a good transparency for further software development. Therefore, the practical aspects of the generated models will be considered and evaluated. This integration model will be applied with an example from the software development at Daimler AG. Finally the integration model will be applied as prototype on internal source code.

Kurzfassung

Die Softwareentwicklung spielt in der Automobilindustrie eine immer wichtigere Rolle. Die steigende Vernetzung zwischen Softwaremodulen und wachsende Anzahl von Softwarekomponenten führt zu hoch komplexen Systemen. Ein modernes "High-End-Auto" besitzt bis zu 100 Millionen Zeilen Code [GTD15]. Um diese Komplexität zu beherschen, wurde die modelbasierte Softwareentwicklung eingeführt. Ein Modell bezeichnet die Abstraktion einer Software von verschiedenen Betrachtungspunkten auf verschiedenen Abstraktionsebenen. Modelle sind in der Softwareentwicklung notwending, um Software zu vereinfachen und weiterzuentwickeln.

Neben der modellbasierten Softwareentwicklung wurde in der Automobilindustrie AUTOSAR eingeführt. Hauptziel von AUTOSAR ist die Verbesserung des Komplexitätsmanagement von Fahrzeugen mit integrierten E/E-Architekturen. Dies wird durch die Wiederverwendung und vereinfachter Austauschbarkeit der Softwaremodule erreicht.

In der Realiät ist die modellbasierte Softwareentwicklung noch nicht durchgehend eingeführt. Die Soll-Prozesse werden nur teilweise getrieben. Das führt unter anderem zu Informationsverlust, Softwarefehler, fehlender Transparenz und dem Risiko an der Komplexität zu scheitern. Das liegt zumeist daran, dass Softwareänderungen direkt im Code implementiert werden, ohne dabei zugehörige Modelle anzupassen. Folglich sind die Software Diagramme inkonsistent mit dem Quellcode.

Diese Thesis fokussiert sich auf die Entwicklung eines Integrationsmodells, welches Reverse Engineering modifiziert, um aus Quellcode Softwaremodelle zu erzeugen. Dabei soll stets der AUTOSAR-Standard berücksichtigt werden. Hauptziel des Integrationsmodell ist es, Architekturmodelle zu erzeugen, welche im Entwicklungsprozess fehlen. Ziel der Modelle ist es, die Lesbarkeit der Software zu verbessern, die Softwarequalität zu erhöhen und Vorteile für die Softwareentwicklung zu gewinnen. Die generierten Modelle sollen Informationserhaltung, Korrektheit und eine gute Transparenz berücksichtigen. Dafür werden vor allem die praktischen Aspekte betrachtet und ausgwertet. Das Integrationsmodell wird abschließend mit einem Quellcode-Beispiel der Softwareentwicklung der Daimler AG angewendet und entsprechend bewertet.

Contents

1. Introduction 1
   1.1. Use of Confidential Data . . . 1
   1.2. Motivation . . . 1
   1.3. Goals . . . 3
   1.4. Thesis Structure . . . 4

2. Foundations 7
   2.1. Model-driven Software Development . . . 7
   2.2. UML Architecture Diagrams . . . 10
   2.3. AUTOSAR Standard used at Daimler AG . . . 11
   2.4. Reverse Engineering of Software Systems . . . 14

3. State of the Art 17
   3.1. Model-driven Software Development with AUTOSAR at Daimler AG . . . 17
   3.2. Reverse Engineering of Source Code and Automotive Systems . . . 19

4. Related Work 21
   4.1. Search Strategy . . . 21
   4.2. Reverse Engineering with AUTOSAR from SW-Component Source code to SW-models . . . 22
   4.3. AUTOSAR based model exchange methodology . . . 22
   4.4. Reverse engineering processing for embedded systems (with AUTOSAR) 23

5. Methodology 25
   5.1. Developing the Integration Model . . . 25

6. Integration Model 31
   6.1. Definitions . . . 31
   6.2. Result . . . 33
   6.3. Development of the Integration Model . . . 34

7. Prototyping 47
7.1. Selection of an Example 47
7.2. Practical Implementation 50
7.3. Result 51

8. Evaluation 53
8.1. Research Questions 53
8.2. Objective Evaluation 54
8.3. Subjective Evaluation 56
8.4. Result 58

9. Conclusion 61

A. Use Case Description and Manual 63

Bibliography 71

List of Figures

2.1. V-Model, own representation based on [Ope05] 8
2.2. UML Class Diagram [RG98] 11
2.3. AUTOSAR: Hardware and software-independent Software [Bun11] . . . 12
2.4. AUTOSAR layered Software Architecture [Tea19b] 13
2.5. Classification Reverse Engineering in V-Model, own representation . . . 15

3.1. V-Model Automotive Development, own representation 18

5.1. BPMN Diagram: Methodology for Practical Development of the Integration Model, own representation with *www.bicdesign-free.com* 26

6.1. Definition of the Integration Model and given tools, own representation 32
6.2. Integration Model, own representation 33
6.3. Module Specific Architecture with internal software modules (data obfuscated), own representation with Rhapsody [Cor08] 35
6.4. Module Specific Architecture for hand-coded software (data obfuscated), own representation with Rhapsody [Cor08] 36
6.5. Diagram for Vehicle-Function-internal Architecture (data obfuscated), own representation with Rhapsody [Cor08] 39
6.6. Diagram for Vehicle-Function-across Architecture (data obfuscated), own representation with Rhapsody [Cor08] 42
6.7. Signal Flow Diagram (data obfuscated), own representation with Rhapsody [Cor08] 44

7.1. "AssistFunction1" Specific Architecture Diagram, own representation with Rhapsody [Cor08] 48
7.2. "AssistFunction1 - AssistFunction2" across Architecture Diagram, own representation with Rhapsody [Cor08] 49
7.3. "AssistFunction1 - AssistFunction2 - AssistFunction3" signal flow Diagram, own representation with Rhapsody [Cor08] 51

8.1. Subjective Evaluation Survey 58

List of Tables

8.1. Objective Evaluation Criteria 54
8.2. Objective Evaluation of Prototyping in 7 55
8.3. Survey probands 56

Chapter 1

Introduction

1.1. Use of Confidential Data

This thesis is written in cooperation with Daimler AG. The practical work is applied based on internal source code. This source code for reverse engineering is developed for new Mercedes-Benz vehicles and runs on a ECU. Thus, the source code belongs to confidential data of Daimler AG. A population of confidential data is not allowed.

A confidentiality declaration is not created because the type of source code does not affect results of the integration model. Relevant is the generation of UML diagrams from source code based on AUTOSAR.

Confidential data in this thesis are obfuscated. This means that all necessary diagram components of models generated by reverse engineering are obfuscated i. e. renamed. Especially chapter 6 and 7 contain results based on confidential source code. These results are obfuscated but base on real data.

1.2. Motivation

The software development in the automotive industry is more important than ever. Due to increasing numbers of control units and software components in vehicles the complexity and connectivity of the entire system is growing. Vehicles become more as "driving software" than "vehicle with software components". Big automotive companies are challenged to master the whole vehicle software. Therefore, the complexity of the software has to be reduced. This can be done by abstractions. In this paper, a model is an abstraction of some aspect of a system. These models show a part of the software as behavior, architecture, structure , requirements or environment etc. in graphical or textual notations. In this paper only graphical models will be considered.

Mastering complex systems requires an efficient and faultless software development process. The model-driven software development is necessary for the modern software development. The processing starts on a high-level abstraction, the requirement analysis, down to system design and software architecture design. Finally the source code is often automatically generated based on the software models. This process will be introduced in detail in section 1.3.

In the automotive industry AUTOSAR as software standardization has been introduced. AUTOSAR simplifies vehicle software by reducing connectivity between the electronic control units (ECUs) and improves the reuse and exchangeability of software components. It was introduced at Daimler consequently in 2011 [SRH+11]. It consists of three main working topics: the architecture, methodology and application interfaces [Tea19b]. AUTOSAR will be introduced in detail in section 1.3.

In the past the software development did not use any standardization like AUTOSAR or models to reduce the complexity. Even though, the end-product, the software for the ECUs, in the automotive industry was always successfully applied in the vehicle. The increasing software complexity claimed the introduction of the model-driven software development process. In reality this process is not introduced at all software engineering departments at Daimler AG. Consistent graphical software models are missing on different abstraction levels. As already mentioned the final software is successfully applied in the vehicle nowadays. Thus, there is no high motivation to recreate all models which are part of the model-driven software development process because the software is running successfully. That is the reason why many models of the model-driven software development are missing. On the one side recreation of missing models would mean a big effort for the developers. But on the other side, the models would yield many advantages like transparency, reuse, documentation or early error detection.

With the aid of reverse engineering these missing models can be reproduced. Reverse engineering will be introduced in section 1.3 more specific. In this bachelors thesis the focus is on reverse engineering with the tool IBM Rational Rhapsody. This thesis will not focus on the generation of textual models but graphical models. The missing models of the model-driven software development process should be selected with consideration of the AUTOSAR-standardization.
These models, reverse engineering and configurations will define the integration model.

1.3. Goals

This thesis aims to improve the software development by replenish parts of the architecture design. Therefore, the software development process at Daimler AG has to be analyzed. After that existing models will be checked to analyze information about the architecture. On the basis of this content AUTOSAR-relevant parts will be elaborated to develop the integration model. To apply reverse engineering in the software development department a reverse engineering process will be developed and documented too.

1.3.1. Analysis of the software development process at Daimler AG:

The first goal in this thesis is to analyze the internal software development process at Daimler AG. In the analysis the software models and the source code will be considered. Therefore, suitable current models which are used for the software development have to be selected. This leads to the first research question:
RQ1: How does the software development process work in the automotive industry (at Daimler AG)?

1.3.2. Development of the integration model:

The development of the integration model contains mainly the improvement of the reverse engineering with the given tool IBM Rational Rhapsody. Therefore, different parts can be customized - e.g. Rhapsody configurations, graphical options, source code architecture, Java APIs and scripts. The goal is to adapt these tools of reverse engineering to get advantages from the models. This lead to the second research question:
RQ2: Do software developer yield advantages from generated model with reverse engineering?

The first step is to select an internal example of software development, in which the architecture diagram is already modeled and source code has been generated automatically. With the aid of this simple example the reverse engineering will be tested in Rhapsody to generate models based on the source code. First results lead to the third research question:
RQ3: Which advantages can be yield using reverse engineering?

The next steps is to constantly continue developing the integration model with software, Runtime Environment and data structure analysis. Therewith, goal is to visualize additional information in a UML architecture diagram. This formulates the last research

question:
RQ4: Can additional information can be visualized with reverse engineering in Rhapsody?

1.3.3. Evaluation of generating software models

In the phase of development of the integration model and after final prototyping the generated models will be evaluated. The evaluation will be applied in two parts. The first part consists of an objective evaluation. Therefore, different criteria of several evaluation methods will be selected to create an individual assessment form. This form is customized for generated models and especially used for the evaluation in the phase of integration model development. The second part is the subjective evaluation as survey. Therefore, different experts will evaluate the quality of generated architecture diagrams.

1.3.4. Prototyping

Here, the goal is to apply practical usage of the integration model for an internal source code example to generate missing software models. The example of an internal source code should be selected in cooperation with software experts. So the final evaluation can be done with practical relevance.

1.4. Thesis Structure

This thesis is structured as following:

Chapter 2 – Foundations: introduces into the foundations of model-driven development, reverse engineering, UML modeling and AUTOSAR.

Chapter 3 – State of the Art: describes the current status of reverse engineering used in practice.

Chapter 4 – Related Work: presents research works for reverse engineering processing and reverse engineering in the automotive industry.

Chapter 5 – Methodology: explains the methodology procedure to develop the integration model and the evaluation.

Chapter 6 – Integration Model: defines development and result of the integration model.

Chapter 7 – Prototyping: describes the practical usage of the integration model with an internal source code example.

Chapter 8 – Evaluation: evaluates the final results. The software models will be evaluated objective and subjective.

Chapter 9 – Conclusion summarizes the thesis and gives an overview of possible future works.

Chapter 2

Foundations

The increasing complexity of today's software forces the software development to start on a high abstraction level. Modern high technology car software contains up to 100 million lines of code [GTD15]. Without any model of a higher abstraction level it would be impossible to control such a big software project. That is why the software development generally includes a development process like the V-Model with different abstraction process steps. To develop the whole software for vehicles top down, from describing the functional behaviour down to generating the code, car manufacturers are already partly automating their software development. On the low-level abstraction the code is already generated automatically from software models [SRH+11].

In this thesis only the verification phase will be considered. Reverse engineering of given source code is applied from coding up to the left side of the V-Model in graphic 2.1. This contains the generation of abstraction of the source code into parts of the module design, architecture design, system design and requirement analysis.

2.1. Model-driven Software Development

This section introduces to the model-driven software development in general along the classical V-Model. After that the software development in the automotive industry will be analyzed. Their differences to classical software development will be evaluated.

2.1.1. Software Development in General

Modern high complex software systems are developed with the aid of a classical V-Model. Independent from agile methods, in general the first steps is the high-level description

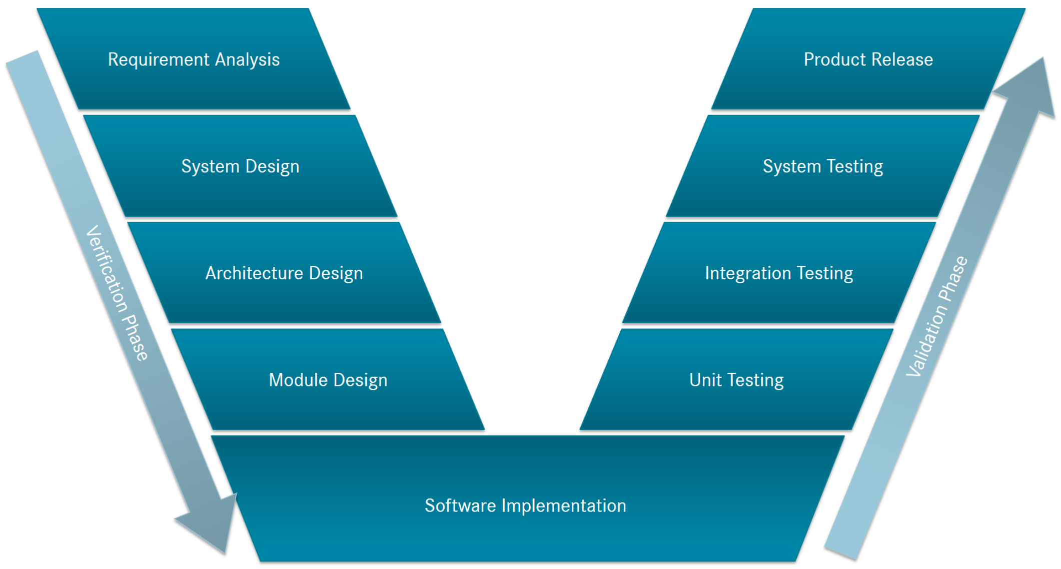


Figure 2.1.: V-Model, own representation based on [Ope05]

down to the software. Starting on defining systems and software tier-structure down to source code implementation. Challenges in the common development are structuring the software clear, creating a great user interface, avoiding bugs and errors etc. The model-driven software development is a assistance proceeding to master all these challenges.

Model-driven software development in this context describes "software development approaches in which abstract models of software systems are created and systematically transformed to concrete implementations" [PBKS07]. The verification phase shown in figure 2.1 is structured as follows (It is important to know that many software development proceedings are varying. In practice the development with the V-Model is often applied in a customized way. This description refers to general proceeding in the software development):

1. **Requirement Analysis:** The requirement analysis describes the software on the most high level abstraction level. The requirements engineer identifies user's needs to define the requirements of the system. User's needs are often documented in a textual form as requirement for the software. These requirements are structured into systems to define the software architecture in the following development steps.
2. **System Design:** The requirements are analyzed and structured to develop the first blueprint of the software. This step often contains system design diagrams to detect logical interfaces between software modules. These diagrams give an overview about the functional architecture, how the software will be structured.

Interfaces between several systems will be detected. This also happens on a high-level abstraction still independent from the hardware realization.

3. **Architecture Design:** The software engineers define software and hardware architecture design based on the system design and requirements. This includes the software tier-structure, hardware component architecture, software structure referred to packages, classes and modules. Furthermore, architecture contains the organization of communication between hardware and software components.
4. **Module Design:** In this phase the software is specified on a low-level abstraction. The developer only works in one system. He considers the software requirements and develops the diagrams based on the defined architecture. here, software modeling happens close to the final code. Architecture diagrams or behavior diagrams describe the software. Often, these diagrams are used to generate the source code automatically.
5. **Software Implementation:** Depending on the code generation this phase includes the whole implementation of the final software. If the code was generated automatically only functions of source code and adjustments have to be implemented. If the code has to be manually written the information from module design has to be implemented.

Usually every process step is executed by an expert for systems engineering, software specification, coding etc. This is why a good collaborative work is necessary for the whole verification software development phase. The knowledge of all involved engineers should help to improve and optimize the final software.

2.1.2. Software development in the Automotive Industry

The specification of V-Model for model-driven software development is a generalization too. The detailed processing depends from department again. Big automotive companies have got several proceedings. This thesis takes focuses on one common wide spread development process.

In contrast to other software systems the automotive software development has different challenges to master. As already mentioned, modern high complex systems have several challenges in the software development. Beyond that, [PBKS07] describes additional challenges of automotive software systems:

- The architecture of electronic control units (ECU) and software components are way more complex. A modern high-end car has more than 100 ECUs. These ECUs always have to communicate. The massive information exchange and high numbers of ECUs lead to a tailored middleware and different more complex software architecture.

- Specific requirements related to reliability, safety and security must be respected. Driving a car is always a security risk for the driver. All car functionalities like break light, warning light, windshield wipers and assistance systems like ABS and ESP have to comply legal requirements. Consequently the reliability and security of software in car is very important.
- The large number of variants and configurations increases the complexity of software. Nowadays Mercedes-Benz has about 30 different models on the market [Hee06]. Variant diversity of cars causes the next challenge to solve. Different car variants need adapted implementation fitted on a specific vehicle.

These challenges lead to an adapted development for vehicle software systems e.g. introduction of AUTOSAR (2.3). High complexity and networking provides well structured development with different abstraction levels. The common development process at Daimler AG is introduced in chapter 3.1.

2.2. UML Architecture Diagrams

The Unified Modeling Language (UML) is a standard language for defining software blueprints. It is a very expressive language, adressing all the views needed to develop and then deploy such systems [BRJ99]. In general there are two diagram types, structural or static diagrams and behavioral or dynamic diagrams. The main vocabulary of UML contains things, relationships and diagrams. The UML is for visualizing, specifying, constructing and documenting the artifacts of a software-intensive system:

- Visualizing: Graphical symbols express architecture and behavior of a software system. To understand an existing software, it is mostly better to get an overview by a graphical model. The beholder will mostly understand the basics of software systems faster and easier.
- Specifying: Building UML models, that are precise and complete, specify the software system for all the design.
- Constructing: Models can specify a software very precise and detailed. The information in models in generated code are often the already specified software. Modeling environments like IBM Rhapsody or Matlab/Simulink are able to construct and generate source code in a programming language automatically from models.

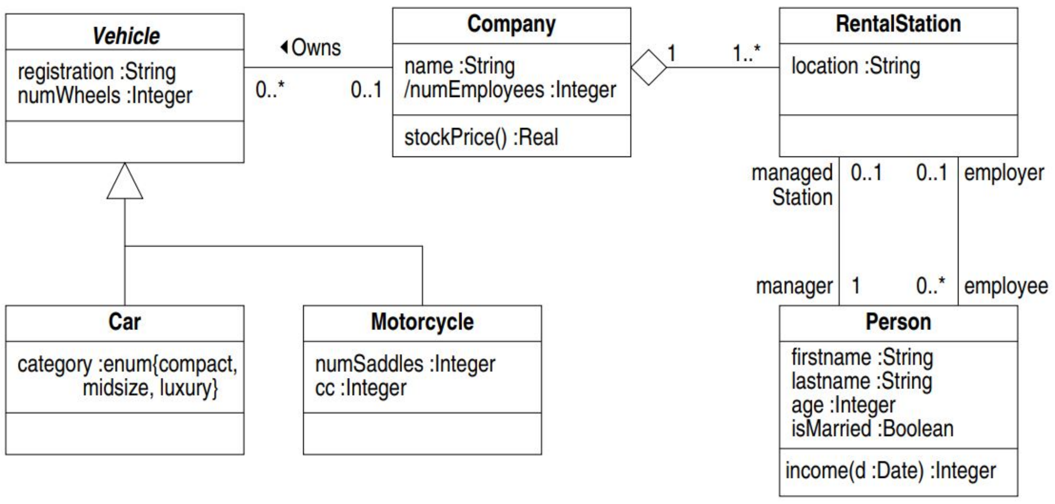


Figure 2.2.: UML Class Diagram [RG98]

- Documenting: A maintained software needs some artifacts in addition to the source code. This includes requirements, architecture, design, project plans, releases and prototypes. UML adresses the documentation of system's architecture and all of its details.

This thesis focuses on the generation of architecture diagrams. They visualize the architecture by using things/blocks [BRJ99] and their relations to each other. Relations can represent a dependency, association, generalization and realization. Figure 2.2 shows a simple example for an UML class diagram. This model shows the high-level architecture of rental company for cars and motorcycles. The relations are specified by an additional documentation. The classes are specified by defined attributes and methods. Later, generated models are constructed similar to figure 2.2.

2.3. AUTOSAR Standard used at Daimler AG

This section introduces to the high-level idea of AUTOSAR (AUTomotive Open System ARchitecture). This thesis requires a basic knowledge about the software architecture of automotive systems. Therefore, the AUTOSAR layered Software Architecture will be introduced in 2.3.2.

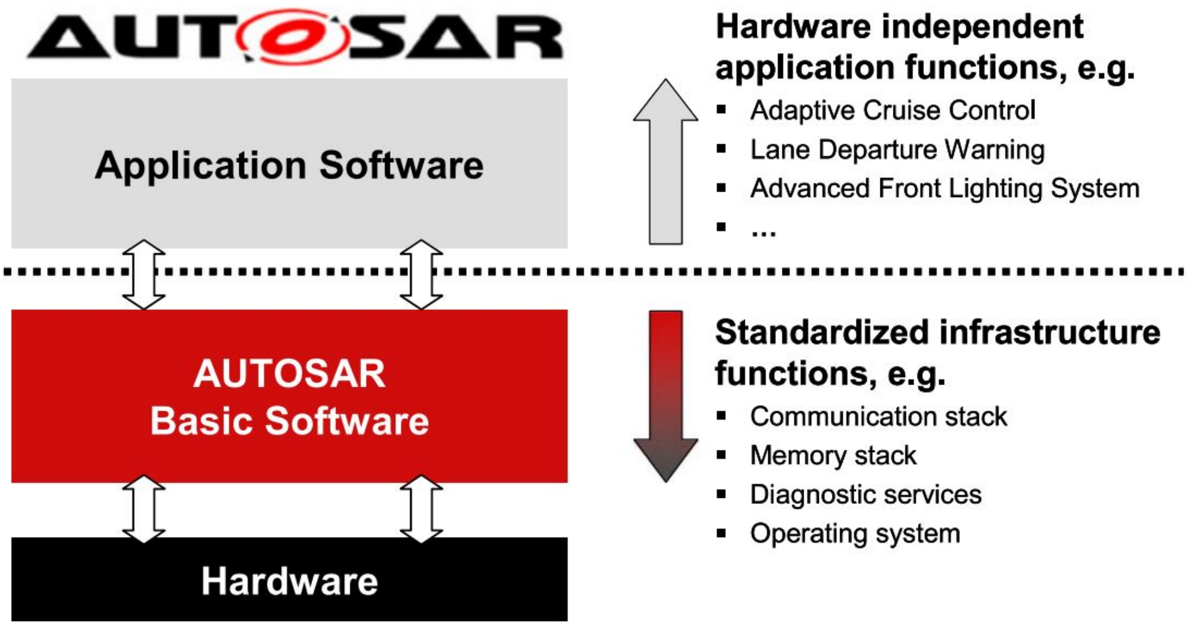


Figure 2.3.: AUTOSAR: Hardware and software-independent Software [Bun11]

2.3.1. AUTOSAR in General

AUTOSAR is a standard for automotive software systems. It has gotten very important for the development of embedded systems. Besides the model-driven software development AUTOSAR is an additional methodology to master challenges in the automotive industry mentioned in 2.1.2. Main goal of AUTOSAR is to improve the complexity management of the vehicles integrated E/E architectures through increased reuse and exchangeability of software modules between Original Equipment Manufactors (OEMs) and suppliers. This is realized by hardware and software-independent software modules. AUTOSAR describes four areas of the automotive software development. The software architecture, methodology, application interfaces and acceptance tests [FMB+09]. This thesis mainly focuses on the first area. The software architecture is the most relevant part for the development of the software models. These four topics of AUTOSAR are defined as following [Tea19b]:

- Software architecture: This part describes the architecture of electronic control units (ECUs) - the AUTOSAR Basic Software.
- Methodology: Definition of exchange formats and description templates to enable a seamless configuration process. This includes software component templates, ECU configurations and the methodology how to use this framework.

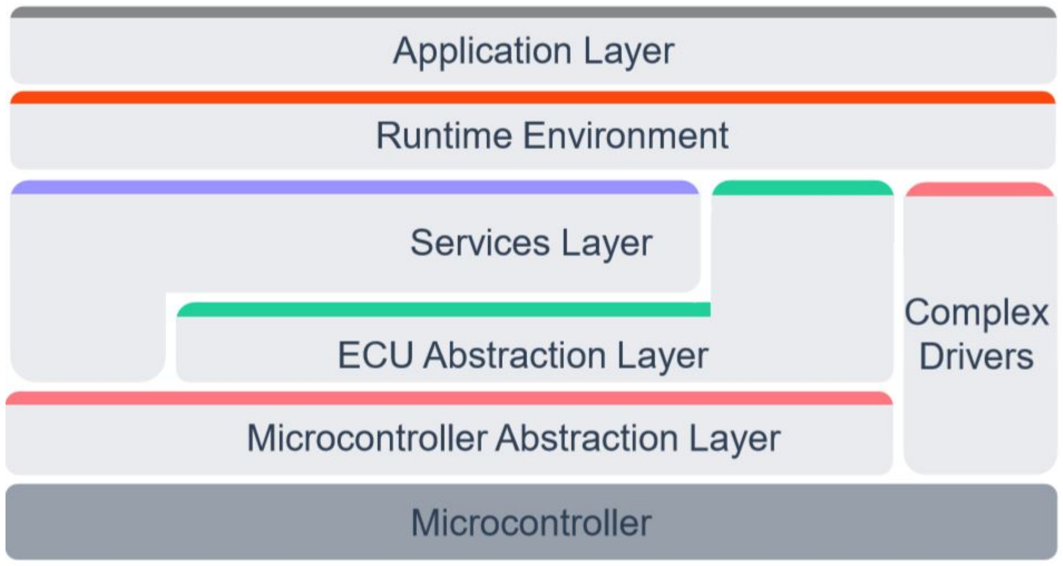


Figure 2.4.: AUTOSAR layered Software Architecture [Tea19b]

- Application interface: Specification of syntax and semantic of interfaces in typical vehicle software systems e.g. Interior, Body and Comfort, Powertrain [MRHK10].
- Acceptance tests: Specification of test cases intending of validate behavior of an AUTOSAR implementation.

Currently Daimler AG is using the "AUTOSAR Classic" release. The current AUTOSAR platform is the "AUTOSAR Adaptive". The main concept of AUTOSAR Classic is the separation of hardware-independent application software and hardware-oriented basic software. The runtime environment (RTE) as virtual bus systems controls the data exchange between the ECUs.

2.3.2. AUTOSAR Classic Software Architecture of Modern Vehicle Systems

The AUTOSAR architecture aims the software and hardware-independent programming. Therefore, three software layers run on a Microcontroller: Application Software, Runtime Environment (RTE) and Basic Software. The AUTOSAR Basic Software is divided into the following layers shown in figure 2.4:

- **Microcontroller Abstraction Layer:** This layer is the lowest software layer of the basic software and makes higher software independent of the microcontroller. The task of the microcontroller abstraction layer is to define an abstraction of the underlying hardware. The goal is to simplify the programming.

- **ECU Abstraction Layer:** This layer is intended to offer the ECU specific services (access to I/O signals). It connects the drivers of Microcontroller Abstraction Layer and makes higher software layers independent of the ECU hardware layout.
- **Service Layer:** The highest layer of the Basic Software offers hardware independent basic services for application. It provides memory services, diagnostic services and ECU state management.
- **Complex Drivers:** The complex drivers layer provides the possibility to integrate special-purpose functionality for instance drivers for devices which are not specified within AUTOSAR. This part includes functionalities that can not be standardized.

The AUTOSAR Runtime Environment abstracts the application layer from the basic software and is responsible for the communication between all software components. The Application Layer contains all software components. These software components are ECU-independent.

In this thesis several software components from the FAP (Driving assistance package) will be used for practical implementation. This package contains functionalities like "Distronic Plus" or "Active Brake Assist". For source code analysis and model generation the Application Layer and RTE will be considered.

2.4. Reverse Engineering of Software Systems

This section introduces reverse engineering. It is necessary to define the term "reverse engineering" in relation to this thesis because there are existing many definitions and use cases.

2.4.1. Definition

Reverse Engineering in general is meant for the recreation of existing systems in higher abstractions. This can refer to generating source code generated from binaries. Here, reverse engineering is applied source code in C for the generation of UML diagrams.

The Institute of Electrical and Electronics Engineers defines reverse engineering as "the process of analyzing a subject system to identify the system's components and their interrelationships and to create representations of the system in another form or at a higher level of abstraction" [CC90], where the "subject system" is the end product of software development. Generally spoken, the output of a reverse engineering activity is synthesized, higher-level information that enables the reverse engineer to better

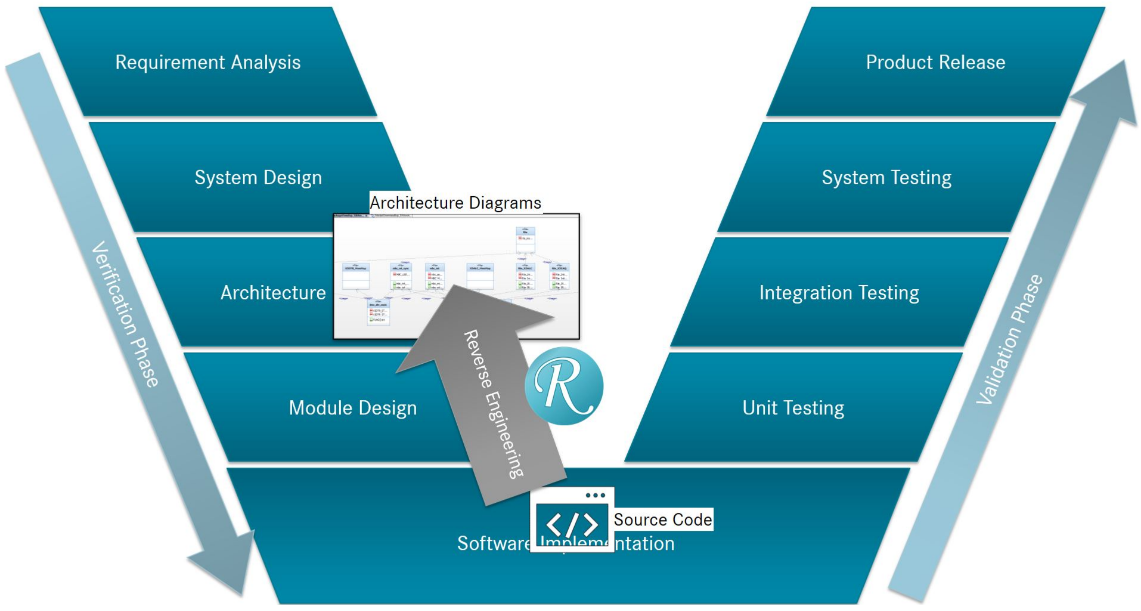


Figure 2.5.: Classification Reverse Engineering in V-Model, own representation

reason about the system and to evolve it in a effective manner. The process of reverse engineering starts with lower levels of information the system's source code, possibly also including the system's build environment [KKM12]. This lower level can obtain programming languages like Java, C/C++, Ada or assembly code.

2.4.2. Reverse Engineering of Source Code

In the context of this thesis, the software of ECUs in vehicles is considered. Models on a higher level are often not complete or inconsistent to source code. Referred to V-Model this means that the coding part is complete. But the steps before the verification phase might be incomplete. To fill these gaps in the requirement analysis, system design, architecture design and module design, reverse engineering is a method to generate models [BLL06]. At Daimler AG the requirements analysis is just textual. That is why this thesis only focuses on graphical models, especially architecture diagrams. Therefore, the verification phase will be passed backwards. The reverse engineering is applied with given technologies i.e. IBM Rational Rhapsody and additional configurations. The goal is to optimize the use of these given technologies to generate a great benefit for the research and development department in software engineering.

Figure 2.5 illustrates the high-level idea of the integration model applied as reverse engineering in this thesis. Input is source code (in C) of a ECU. Output are diagrams visualized with Rhapsody, classified in the architecture design of the V-Model.

IBM Rational Rhapsody provides reverse engineering [Cor08] of source code. Rhapsody is a modeling environment based on UML for system engineers and software developers. It is able to generate UML and SysML models based on source code in C++, C, Java and Ada. The user has the opportunity to have a graphical view on the source code from different viewpoints. This tools reverse engineering function will be modified to generate useful models for the development.

Chapter 3

State of the Art

The following section describes the usual case of model-driven software development with AUTOSAR at Daimler AG of today. In practice there are many variations of the model-driven software development process depending on application area. The selected example in C-Code was developed with the common development process in section 3.1. 3.2 introduces state of the art of reverse engineering used in the automotive industry.

3.1. Model-driven Software Development with AUTOSAR at Daimler AG

The model-driven software development with AUTOSAR was introduced on a larger scale the first time in the early 2000's [SC08] (Development of 204 series, C-Class). Since then, the model-driven software development is used in almost all areas of software development at Daimler AG.

In detail internal software development processes are partly different from each other. This depends on the usage of the software. For example a control software needs to simulate the software early to detect error and evaluate the behavior. This is why behavior models like sequence diagrams are very important for control software. A trip computer in vehicles has to collect many information to show them the driver. That is why the ECU of trip computer has got many interfaces to other ECUs. In this case a component diagram is very useful to detect all interfaces.

Figure 3.1 illustrates all steps of the verification phase and their content:

1. The requirements engineering is applied textual mostly in a tabular form. Therefore, the tool IBM DOORS [SRH+11] is used.

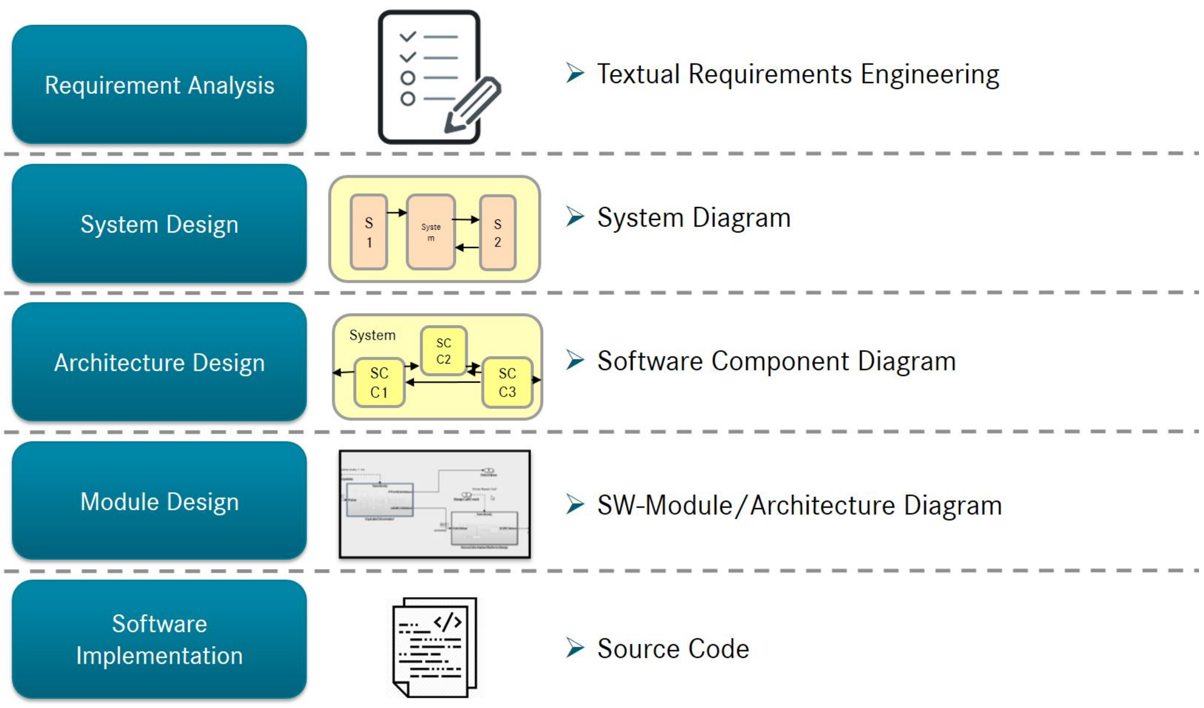


Figure 3.1.: V-Model Automotive Development, own representation

2. The system diagram divides the whole vehicle software into logical systems. This separation provides a good collaboration and an efficient development. The created diagram therefore shows the architecture of vehicles software. Interfaces and information exchanges become clear.
3. The architecture design defines the software and hardware architecture. The hardware architecture with a high number of ECUs and high connectivity have to be well structured. Furthermore, software has to be separated into logical useful packages and modules. The communication and dependencies between several modules is specified.
4. The module design depends on the use-case of the software module. In some cases behavior diagrams are useful, in other cases another low-level architecture diagram is practicable. These diagrams often use an automated source code generation to implement the final software code consistently.
5. The final implementation is in the respective programming language, mostly C or C++. Partly source code is automatically generated by modeling tools. Other parts are still manually implemented.

As already mentioned usually three kind of models are used in the software development at Daimler. The first one, a system diagram, describes systems interfaces. It is similar to a simple UML class diagram. The second one is the software component diagram. It describes the architecture of a system consisting of their software components. In the module design phase the behavior of the ECUs is modeled by a behavior diagram. Usually in the software development these diagrams are similar to the activity or state-chart diagrams. In the automotive industry Simulink (Targetlink) is often used for modeling and simulating the behavior of ECUs.

3.2. Reverse Engineering of Source Code and Automotive Systems

In fact, there is no widespread or popular practical deployment of automated reverse engineering in the automotive industry at Daimler AG. There are different approaches of reverse engineering in the general automotive industry. To generate UML class diagrams based on source code there are different tools and approaches.

3.2.1. Reverse Engineering Tools Benchmark

In [CN15] different technologies of reverse engineering approaches are compared. Various tools like IBM Rational Rhapsody, ArgoUML, StarUML, etc. are compared. All tools are analyzed in terms of complexity, performance and effectiveness. These tools represent the modern state of the art of reverse engineering. The comparison consists of a comprehensive set of Java-based targets for reverse engineering and a formal set of performance measures with which tools are analyzed. With a look on the results of this benchmark it is notable that Rational Rhapsody is the only tool which detects all software elements with 100% reliability. Consequently Rational Rhapsody is the most suitable (2015) tool for reverse engineering of architecture diagrams according to [CN15].

3.2.2. Reverse Engineering of automotive embedded Systems

There are two approaches of reverse engineering for the automotive industry based on source code. The first approach is about reverse engineering of ECUs. [HMSN10] introduces a concept to generate models based on source code of ECUs and ensure safety regulations. This requirement enables the reuse of ECU software. Therefore different reverse engineering algorithms are compared and evaluated. In theory this approach ensures the reuse and safety regulations of models generated by reverse engineering.

Therefore, several reverse engineering algorithms are introduced. They provide to generate models from source code, which can be used for further development. But for productive usage there is no wide spread tool introduced yet.

3.2.3. Reverse Engineering of CAN Bus Systems

The second approach focuses on reverse engineering of CAN (Controller Area Network) Buses to learn about the communication between the ECUs. [FAK11] and [HVB+17] describe two approaches for techniques which analyze communication between ECUs via CAN bus systems. Therefore, the data of CAN bus provide a basis to detect hardware oriented measurement variables like deviations or flow rates of liquids. Both papers show that reverse engineering is practicable and useful for some use-cases.

However this thesis only focuses on reverse engineering of vehicle software systems. Results of reverse engineering should support further software development by generating architecture diagrams.

Chapter 4

Related Work

This chapter introduces to related work of this thesis. Firstly, the search strategy is presented in 4.1. Secondly relevant related work are proposed in section 4.2. In section 4.2 the theoretical aspect and correctness of reverse engineering is defined. After that AUTOSAR based model exchange methodology, which can be used for reverse engineering 4.3, is introduced. Finally the description of reverse engineering processing for embedded systems especially with AUTOSAR standard is described in 4.4.

4.1. Search Strategy

The "Schneeball"-method of [BM16] is used as search strategy. The advantage is that the author does not have to know about all basics of the topic in the beginning. It starts by researching for general information about the topic of the thesis to get a basic knowledge.

The basics in this work contain AUTOSAR, model-driven software development and reverse engineering. Starting from general paper the goal is to get an overall view of the basics first. From there, the literature research goes into more deep, relevant information by researching for referenced literature. To look for more deep information the search was focused on more specific papers: Research paper of model-driven software development processing at Daimler AG (with AUTOSAR) and reverse engineering of AUTOSAR software has to be analyzed.

The order of searching for related work was structured from internal to external. This means the search started in a Daimler internal database with scientific works and other documents written by employees. This also includes interviewing the experts with regard to software development with AUTOSAR. After the internal search, open platforms especially *Google Scholar* was used to find related work. Therefore, all combinations of several keywords were used e.g. reverse engineering, model driven, AUTOSAR, reverse engineering AUTOSAR, model driven AUTOSAR, etc.

Considering the software development process at Daimler AG and reverse engineering for these models the most relevant related work for this thesis can be selected and classified as following:

- Reverse Engineering with AUTOSAR from SW-Component source code to SW-models
- AUTOSAR based model exchange methodology
- Reverse engineering processing for embedded systems (with AUTOSAR)

4.2. Reverse Engineering with AUTOSAR from SW-Component Source code to SW-models

This scientific work [KKM12] focuses theoretical aspects of reverse engineering with AUTOSAR. The goal is to apply reverse engineering algorithms on old components which were not developed with model-driven development. Thus, the development of new components will be accelerated. With the aid of generated models from old software components, new software should be developed. The source code in vehicles is often placed in security critical systems like the airbag function. In this systems the quality standard is very high. Thus, it is helpful to transform existing models with the aid of reverse engineering algorithms to test certain security properties.

For recreating models with reverse engineering algorithms certain properties have to be fulfilled to guarantee correctness. Section 2.3 of [KKM12] is mostly relevant for this thesis to choose the right examples of source code for reverse engineering. The described properties have to be ensured for guaranteeing the correctness of generated software models in this work. Considering modern software standards, these security properties are ensured for in this thesis used source code example. This work will not analyze theoretical algorithms behind tool Rhapsody. The focus is on the practical generation of the generated models with given technologies and adaptions.

4.3. AUTOSAR based model exchange methodology

In [PB06] Pagel and Broerkens evaluate XML exchange formats, which allow wide configuration possibilities based on AUTOSAR. Therefore platform independent UML model transformations to XML-schemes are analyzed. With the aid of these XML-exchange formats the reverse engineering is easy to realize for generating UML-diagrams compliant with AUTOSAR methodology. These XML schemes define the software architecture and model transformations. First XML exchange files are defined to describe the software models with AUTOSAR standard. This is the most important and relevant part for this thesis. With the aid of description and current state of the AUTOSAR standard release [Tea19a], relevant parts of the AUTOSAR standard can be considered for the model generation.

In [GHN10] one approach for a model synchronization between software and system models are introduced. This bachelors thesis does not consider model transformations or synchronizations. Only one way, from source code to models is applied. For future works, introduced in chapter 9, this paper could be important.

4.4. Reverse engineering processing for embedded systems (with AUTOSAR)

The integration model, developed in this work, should contain a reverse engineering process to apply it in practice on several source code examples. [KKM12] focuses on tools and techniques for the software reverse engineering in the domain of complex embedded systems. The workflow consists of three steps: Extraction, analysis and visualization. For each step IBM Rational Rhapsody can be used as tool in this work. [KKM12] especially introduces an approach for timing analyses in behavior models generated via reverse engineering. But this thesis will not consider behavior models. The reverse engineering architecture will support and be part for development of the integration model.

[Sai14] introduces a interactively reverse engineering process partitioned in three steps. The generated models are based on the AUTOSAR standard and used for real-time systems. It mainly aims the observation of temporal behavior. This aim is not as the same as in this thesis. But the introduced three steps especially step 2 in addition to [KKM12] will support the process for developing the integration model.

In contrast to the related work this aims a widely usable process of reverse engineering considered by the AUTOSAR standard to improve the further software development in practice.

Chapter 5

Methodology

This chapter introduces the methodology of developing the whole integration model. Therefore, two scientific work for reverse engineering [KKM12] and software development process [Rup10] are adapted. The procedure is based on a cycle of continuous developing.

5.1. Developing the Integration Model

This section introduces the methodology of developing the reverse engineering part of the integration model. The process is shown in 5.1. Developing is done by iterating the process several times, taking customization in every iteration and improve the quality and benefits of generated diagrams. It is based on Royce's waterfall model with iterative feedback [Rup10]. The "Requirements" and "Design"-step of Royce's waterfall model in this thesis is realized in the "Objectives"-step. Furthermore, high-level reverse engineering process workflow is introduced in [KKM12]. These three main steps are adapted to this thesis (based on [KKM12] Fig. 1):

- Extract: Source code, RTE and build scripts extractions of FAP 5 $\rightarrow$ corresponds to *Start*, not included for continuous development
- Analyze: Analysis of Fact Extractors $\rightarrow$ corresponds to *Development Integration Model 5.1.4*
- Visualize: Generation of architecture diagrams $\rightarrow$ corresponds to *Practical Implementation 5.1.2*

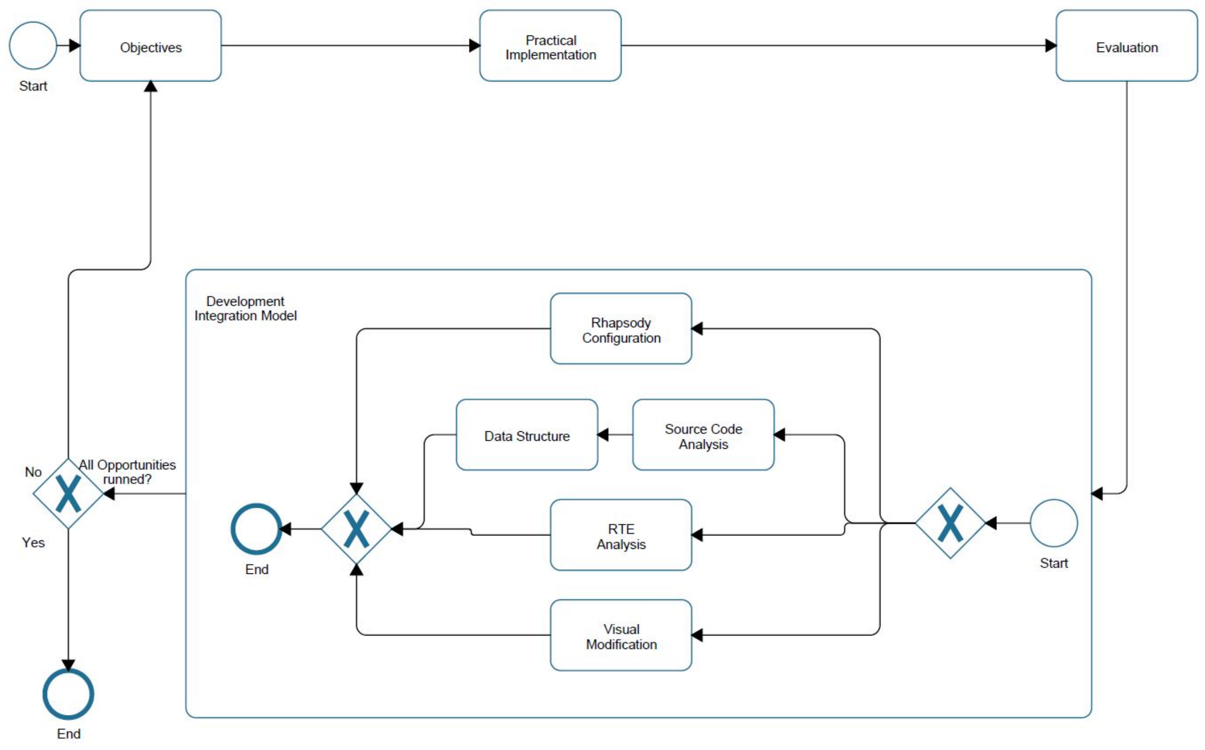


Figure 5.1.: BPMN Diagram: Methodology for Practical Development of the Integration Model, own representation with *www.bicdesign-free.com*

5.1.1. Objectives

First, objectives are defined. In contrast to other reverse engineering tools, IBM Rational Rhapsody provides the best quality of generated models [CN15] (2015). That is why the first practical implementation is done without any adaptions and usable results can be expected.

5.1.2. Practical Implementation

The second step includes the practical implementation multiple times with different inputs. This refers to different source code and various numbers of source code files as input for reverse engineering with Rhapsody. The practical implementation is done multiple times for quality assurance. By generating diagrams for once, it is not ensured that the quality of these diagrams stay consistent for other inputs. Different source code files and a different size of projects still cause errors, wrong or missing information. To minimize this risk, the practical implementation is done multiple times with different size of projects and different source code files. Thus, the statement of the evaluation is more expressive.

5.1.3. Evaluation

To recap the main goal of this thesis 1.3: Development of an integration model for generating software models by reverse engineering existing source code so that the software developer gets benefits. Goal of the evaluation is to have a review, reflect about the results and compare it to objectives. After evaluating, potentials for improvement should be detected and implement in the next iteration. To detect potentials for improvement the evaluation is separated into three parts:

- Quality Evaluation
- Objective graphical Evaluation
- Subjective Evaluation

Evaluation is done in every iteration of the process. In the last iteration, the prototyping, the evaluation is done extensively. Otherwise evaluation is carried out on a smaller scale. Depending on the adaption in previous iteration the evaluation proceeds individual. Only influenceable parameters of diagrams are evaluated. Reverse engineering algorithms of Rhapsody are not rated. As already known, IBM Rational Rhapsodys reverse engineering algorithms perform stable [CN15] compared to the technological quality. So, it is assumed that the diagrams represent the full information content.

In the final evaluation in 8, the quality rating is realized by objective and subjective evaluation. The quality evaluation is done to compare intermediate results with its objectives.

Quality Evaluation

The quality evaluation is necessary in every intermediate iteration process because results in each iteration are different. This part focuses on comparing to objectives. The goal is to detect further potentials for improvement. Main questions for this evaluation are always:

- "Does software developer get benefits with the aid of generated diagrams?"
- "Were the objectives achieved?"
- "Where is still potential for improvement?"

To answer these questions several criteria are selected. These criteria are partly taken by [SW05] the benefit. A benefit can be e.g. detection of new interfaces, detection of unnecessary interfaces, graphical representation of the architecture or detection of dependencies from different viewpoints. Chapter 8 introduces to the selected criteria of final evaluation.

Objective Graphical Evaluation

The goal of this part is to measure the graphical representation of generated models. The readability of software diagrams strongly depends on graphical parameters like symmetric arrangement, hierarchical order or a suitable amount of information. That is why graphical criteria are evaluated, separated into the following cluster:

- Perceptual Organization
- Perceptual Segregation
- Cognitive Effectiveness

These are chosen by a set of [MH08] and [SW05]. The objective graphical evaluation is done in the first iteration to rate default design configurations of Rational Rhapsody. In intermediate steps these criteria are only rated on high-level, if relevant. Lastly the graphics are evaluated in detail in 8 .

Subjective Evaluation

At the end the integration model will support software developers. Thus, opinions of end users are important as feedback. This is why the quality and graphical part of generated models are evaluated by humans. This is done in every iteration by a group of humans (1-3 persons) of a small setting. In the end subjective evaluation is done by are larger group of humans (8 persons). Therefore, different criteria are evaluated:

- Visual Presentation
- Diagram Content
- Source Code Readability
- Improvement On-Boarding
- Improvement Software Quality

For a benchmark several end users will be asked about the difference of understanding the source code with and without the corresponding software models. In chapter 8 specific criteria are introduced to take part in a small survey. In intermediate steps, these criteria are only evaluated on high-level. The goal of this part is to measure all parameters from objective, quality and graphical evaluation on a subjective level.

5.1.4. Development of the Integration Model:

This phase can be executed in different ways. There are several parameters which can be modified to improve the quality of generated models. Depending on objectives and evaluation, it is reasonable to adapt different parameters. For example, it is suitable to configure the design properties in Rhapsody to reach a clear readability. More accurate, in the context of this thesis there are four parameters to adapt for improving the integration model:

- **Rhapsody Configuration:** Pre-processor options of reverse engineering in Rhapsody can be configured. Therefore, *keywords, defines and undefines* can be modified before importing C-Code-files.
- **Data Structure:** The data structure of the whole software project has also impact over generated models. By modifying several source code files into different hierarchical level, more dependencies and interfaces can be detected.
- **RTE Analysis:** The RTE (Runtime Environment) is responsible for the whole communication between ECUs and software modules. The communication is mapped in an adaption XML-files. Thus, all these files can be analyzed to detect interfaces and communication flows.
- **Visual Modification:** Rhapsody allows to configure the visual representation of software models. With it, visual representation can be improved.

The decision what to adapt in the respective iteration, is influenced by different factors. As already mentioned, it is important to reflect about objectives in the evaluation. Here, potentials for improvement can be detected. Furthermore, before acting, the effort is estimated. In cooperation with experts from IBM and software developers from Daimler AG, efforts can be estimated realistically. Lastly the expected results can be estimated too. Based on these criteria a certain "path" in figure 5.1 for one iteration is selected.

Chapter 6

Integration Model

This chapter introduces the integration model developed with the methodology presented in chapter 5. First the term "Integration Model" in this context is defined. After that the result is presented. At last the development of integration model is described.

The integration model takes a specific source code running on a ECU as input. The source code is implemented on a ECU in modern Mercedes-Benz-cars for the functionality of advanced driver-assistance systems (FAP 5). Well-known functions like "DISTRONIC" or "Lane departure warning system" are implemented on this ECU. All diagrams, shown in this section, are based on these confidential data. Thus, names of model elements are obfuscated.

6.1. Definitions

6.1.1. Integration Model

The term integration model can be used in different contexts. In this thesis the integration model is realized by a Rhapsody Profile to integrate source code from the software implementation phase (2.1.1) into architecture design phase. It contains the functionality of reverse engineering from Rhapsody included by extended functions and precomputations. Output are UML architecture diagrams which show dependencies between modules.

To define and develop the integration model, illustrated in figure 6.1, different tools are given:

- Rhapsody Reverse Engineering Configurations

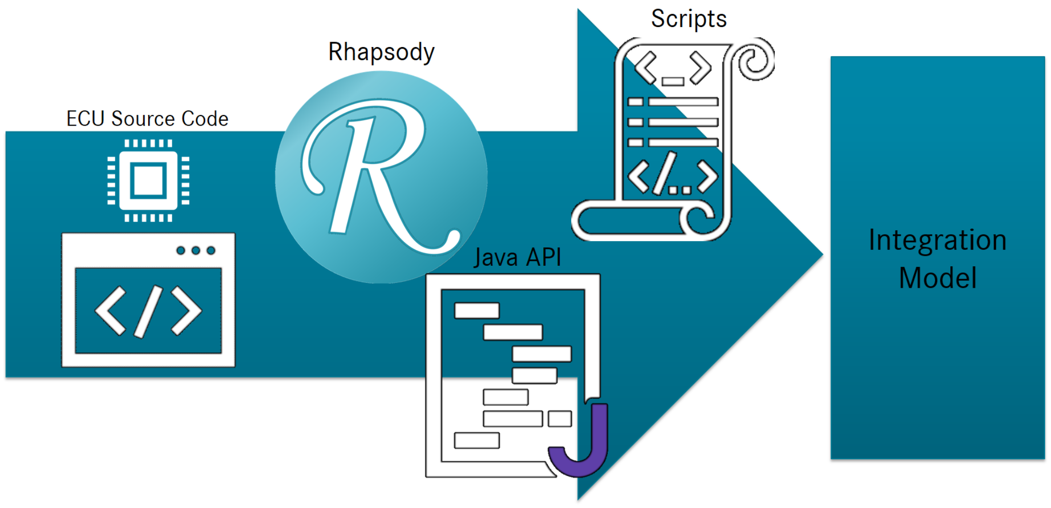


Figure 6.1.: Definition of the Integration Model and given tools, own representation

- Rhapsody Display Options
- Scripts
- Java APIs

For the final result, everything is included in a Rhapsody profile which offers reverse engineering functions adapted for the given source code. This is done by a helper-file which defines includes of JAR- and batch-runnable files. All functions can be called in Rhapsody by clicking on certain buttons.

6.1.2. Vehicle-Function

As already mentioned, the source code implements different vehicle functions. A vehicle function describes a customer-experienceable or -executable function e.g. DISTRONIC or break assistance. Each package of the source code corresponds to a vehicle function.

6.1.3. Software Module

Each package of the source code consists of several functional clustered software modules. Each module is realized by a C-Code and header file. The UML architecture diagrams, shown later, consists of blocks corresponding to software modules.

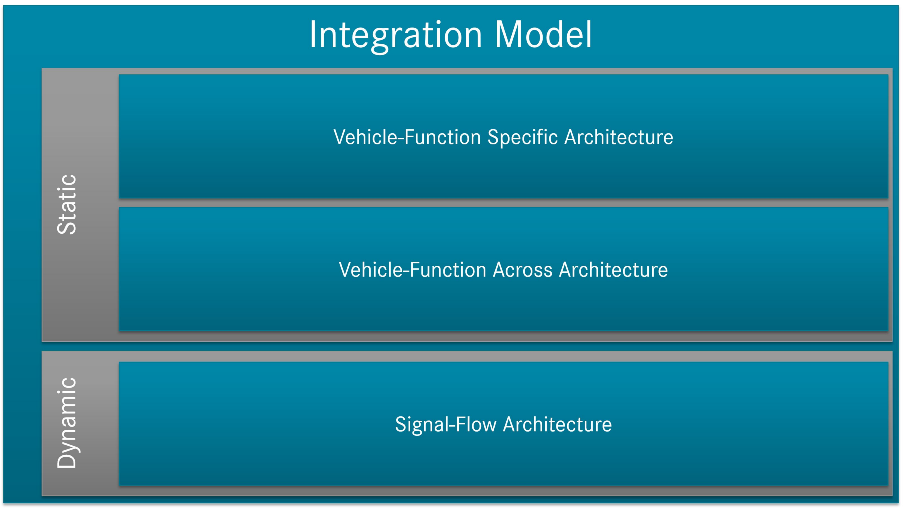

Integration Model
Static
Vehicle-Function Specific Architecture
Vehicle-Function Across Architecture
Dynamic
Signal-Flow Architecture

Figure 6.2.: Integration Model, own representation

6.2. Result

The final integration model consists of three functionalities, illustrated and clustered in 6.2. The vehicle-function specific and vehicle-function across architecture show static dependencies for one or multiple vehicle-functions. The signal-flow architecture visualizes potentials for dynamic dependencies. These dependencies were analyzed in a signal flow analysis. All functions are implemented as scripts and Java API for Rhapsody. The Profile "ExtendedReverseEngineeringAutomotiveC" contains these three functions in Rhapsody named as:

- Vehicle-Function specific Architecture Diagram
- Vehicle-Function across Architecture Diagram
- Signal Flow Dependency Architecture Diagram

The Profile is loadable into a Rhapsody Project area, so that the integration model is usable for every Rhapsody user at Daimler AG. The usage of these three functionalities is described in a use case document of the Project SEED (Systems Engineering Enhancement @ Daimler) and attached to appendix A. Furthermore, a feature document is written. It describes source code of the Java API and the scripts for further development. This is necessary for example adapting the integration model for other ECUs.

6.3. Development of the Integration Model

In chapter 5 the methodology to improve the integration model is introduced. This process is iterated four times. This chapter loops four times through the development process. Every step introduces the procedure and presents intermediate results.

6.3.1. First Iteration

Objective

Main goal is to generate architecture diagrams which improve the source code readability, software quality and yield advantages for software development. This leads first to the following objectives.

Generation of architecture diagrams with basic reverse engineering: First, no generated models based on this software, were known. Thus, no context between diagrams and source code could be analyzed. The first goal is to test basic reverse engineering with Rhapsody and generate first models by importing the source code. Furthermore, the diagrams are analyzed to detect represented and missing information.

Practical Implementation

A part of first practical implementation results are shown in figure 6.3. As already mentioned in 1.1, all data are obfuscated but based on real data.

Figure 6.3 shows one diagram from the first iteration. Here, reverse engineering was applied with default configuration without any additional modification. The software structure with functions, packages and modules was imported, so that for every vehicle-function an architecture diagram was created. Figure 6.3 shows an example for the architecture of one vehicle-function. One block represents one software module. Each association represents a dependency between two modules. The relation from "main" to "module3" means that "main" depends on and includes "module3". It is noticeable that the software modules are hierarchical ordered into levels. The top level software module is responsible for all signals and variables. The second level module provides functions for the third level which contains the whole computation. The bottom level, the main module, calls all modules above and initializes them.

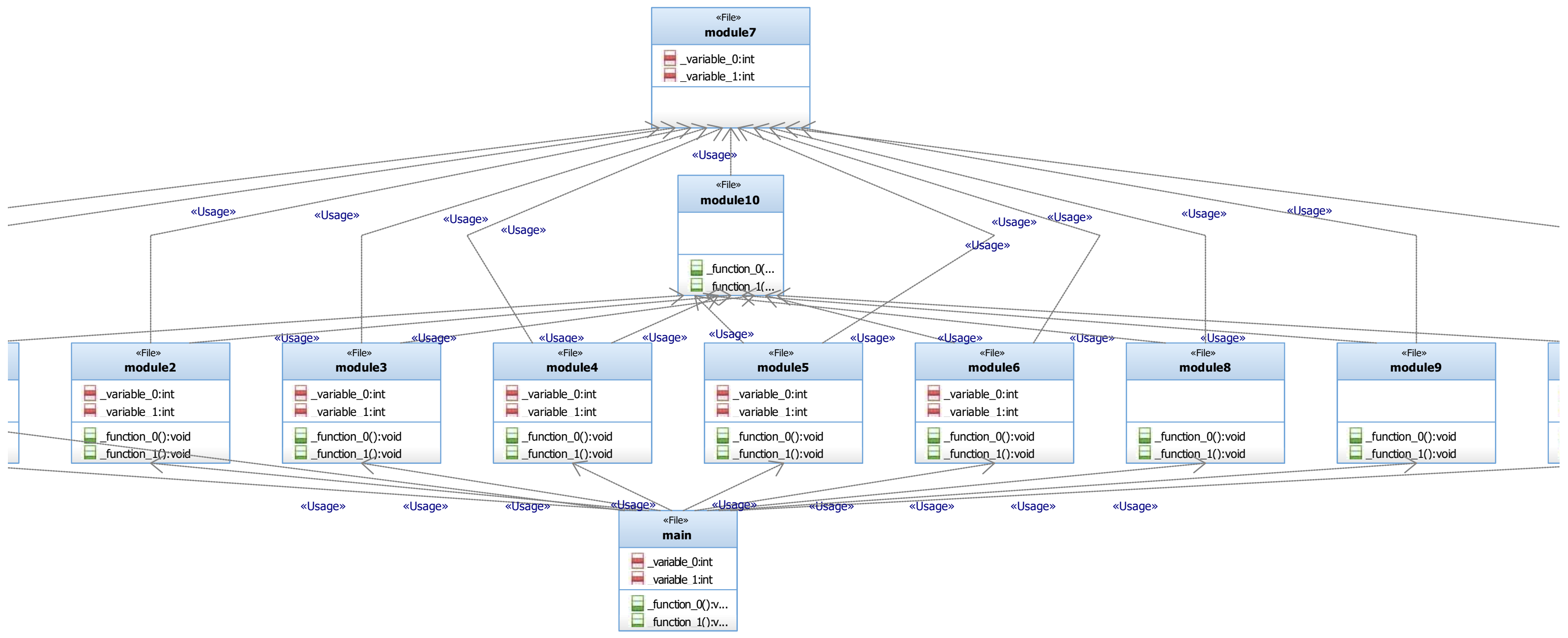


Figure 6.3.: Module Specific Architecture with internal software modules (data obfuscated), own representation with Rhapsody [Cor08]

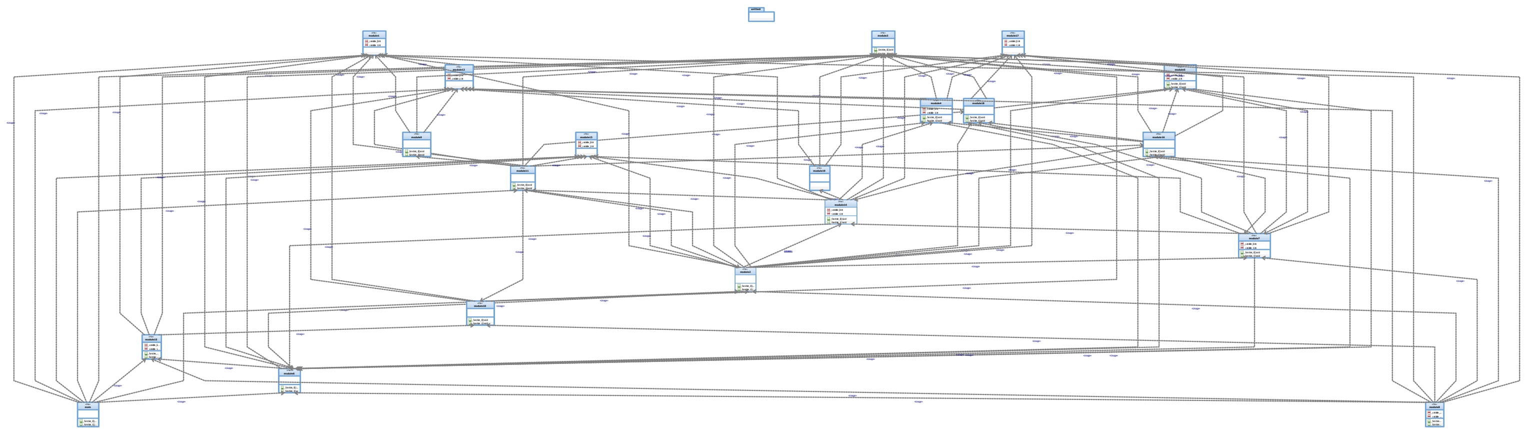


Figure 6.4.: Module Specific Architecture for hand-coded software (data obfuscated), own representation with Rhapsody [Cor08]

Most of the diagrams like 6.3 are based on Matlab/Simulink auto generated source code. Thus, the software architecture looks well structured. The generation automatism causes a consistent and clear structure. Furthermore, a small part of the software is hand-coded (manually written), e.g. figure 6.4. This model should not visualize any content about the source code. It shows the difference of complexity between figure 6.3 and 6.4. This part differs a lot from auto generated code. Figure 6.4 looks way more unstructured and contains a lot more software modules and dependencies.

Evaluation

The evaluation of intermediate results contains only small parts of the overall evaluation in chapter 8. For each iteration only relevant criteria from 8 are considered. Basic reverse engineering functions should be tested, how close main goals of this thesis 1.3 can be achieved.

Within the module hierarchy, first architecture diagrams could be created successfully. Dependencies and inheritances are visualized. The software module hierarchy is plain to see and the perceptual organization is clear. Viewer of these models will definitely get a better overview of the source code from one vehicle function than only reading the code. However, the diagram must be viewed critically with regard to its content. Each diagram for a vehicle-function just shows the architecture of internal software modules. Later it is analyzed that the content is not complete.

In contrast to auto generated code models in 6.3, a hand-coded software diagram is shown in figure 6.4. These diagrams do not give a good overview with clear structure. Figure 6.4 contains confusingly many model elements. Caused by lots of dependencies on different hierarchy levels, this diagram is very extensive. With the aid of these diagrams, it is recognizable that auto-generated code is better cleaned up and structured.

It can be concluded that software architecture of manually written source code should be tide up and better structured. The hand-coded source code is only a small part of the vehicle-functions. Because there is no clear advantage of hand-coded diagrams, this part will not be considered in this thesis anymore.

Development of the Integration Model

As already mentioned, diagram 6.3 is not complete regard to the visualized content. A first source code analysis shows that Rhapsody draws dependencies of C-Code for included files. But the generated model in 6.3 only shows included files of the same vehicle-function. After analyzing the data structure of the software, it can be held, that package-across dependencies are not visualized. Even one abstraction level higher, the architecture diagram on package level, shows no direct dependency between packages.

For this reason a script is written for preparation. This Batch-script restructures the software, so that diagrams for vehicle-functions show the complete structure and dependencies. The batch-script in combination with reverse engineering of Rhapsody build the first functionality of the integration model.

6.3.2. Second Iteration

Objective

Generation of vehicle-function-specific architecture diagrams: After creating first models, second goal is to generate vehicle-function-specific architecture diagrams. This means that generated diagrams should represent the software architecture of one vehicle function 6.3 and additionally relations to external software modules. Here, external modules refers to not-vehicle-function package internal ones(the whole source code of the ECU). That is why the second goal is to set one vehicle-function as viewpoint and generate vehicle-function specific architecture diagrams supported by a script.

Practical Implementation

Figure 6.5 represents an example for a module-specific architecture diagram for a vehicle-function. In contrast to 6.3, figure 6.5 shows additionally external module dependencies. External modules are named as "ext_module". Setting the vehicle-function as viewpoint, figure 6.5 shows all dependencies in the whole software.

Here, reverse engineering was applied with a prerunning script. This script restructures the software. External software modules from the vehicle function were analyzed and added to the model. These included modules are searched in the whole software. Thus, all dependencies to internal and external modules are shown in 6.5.

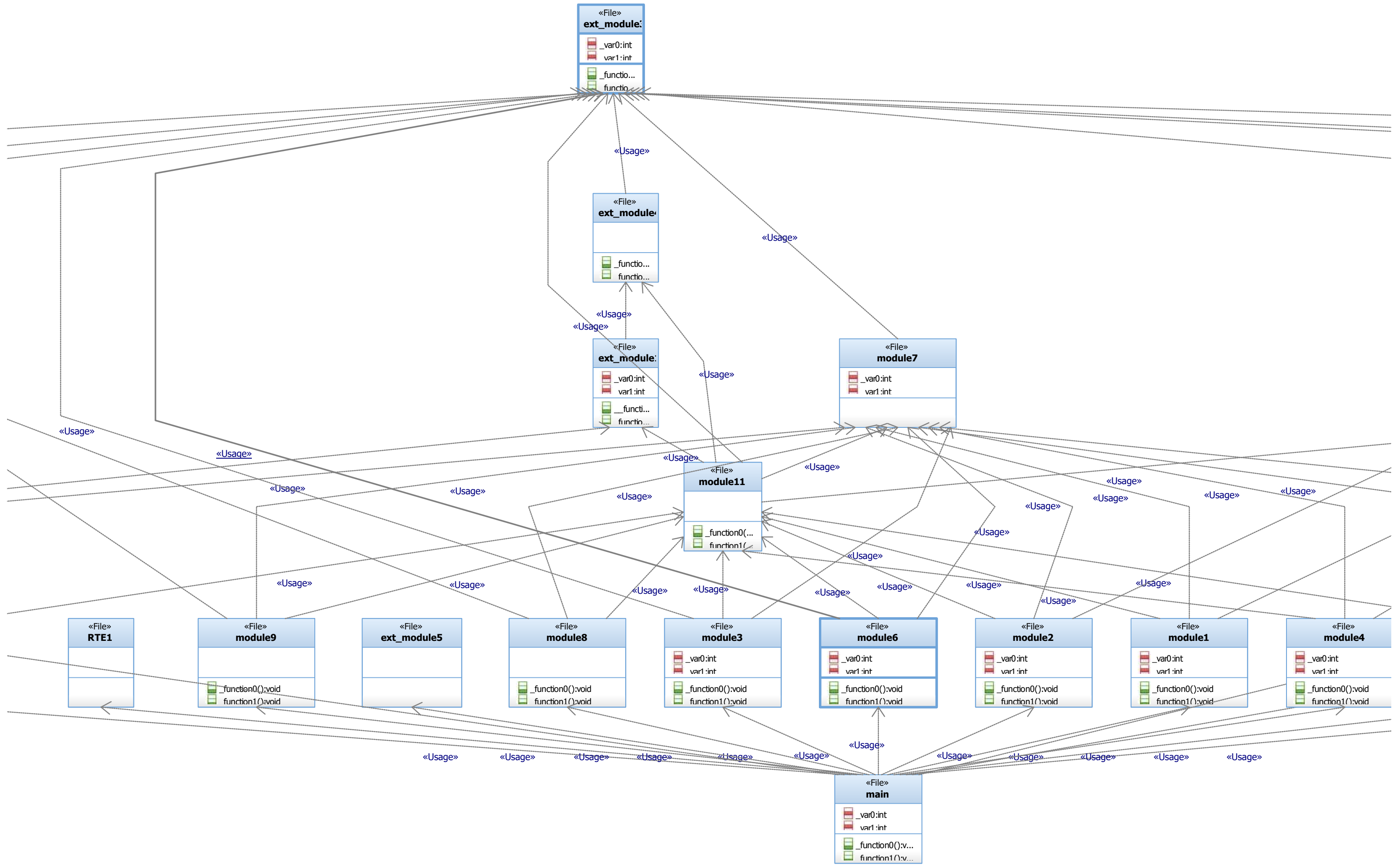


Figure 6.5.: Diagram for Vehicle-Function-internal Architecture (data obfuscated), own representation with Rhapsody [Cor08]

Evaluation

In the second iteration, goal was it to generate vehicle-function-specific architecture diagrams with additional external software modules. By analyzing several diagrams and the corresponding source code, it can be observed that the diagrams are complete. So, every dependency to external modules are displayed. In terms of information content, the goal is achieved. Regard to visual representation, the diagrams are well structured again. Hierarchy and dependencies are clear.

Observing all external software modules it can be observed that no module is in relation to another vehicle-function. This fact leads to the question, how two vehicle function are in relation.

Development of the Integration Model

First, the functionality of script 6.3.1 for software structure preparation was tried to integrate in Rhapsody. Therewith the functionality would be more homogeneous and better integrated into Rhapsody. Therefore, properties and reverse engineering configurations can be adapted. After some attempts, it came to the result, that this functionality can not be caught by properties or reverse engineering configurations. That is why the script is necessary. To integrate the functionality into Rhapsody, the script is callable by an integrated tool-function with "Tools > Automotive Reverse Engineering > Preparation Module Specific Architecture Diagram".

In the next step, to generate a architecture diagram for multiple vehicle-functions, software structure and source code were analyzed again. First intermediate results show that no dependency between software modules of vehicle functions are visualized. After analyzing source code files indirect dependencies via external software modules could be detected. Based on this a second script was written. This script analyzes dependencies of multiple vehicle-functions and restructures the source code files. This functionality should run as preparation for reverse engineering again. It represents the second function of the integration model with static source code analysis: "Preparation Module Across Architecture Diagram".

6.3.3. Third Iteration

Objective

Generation of function-across architecture diagrams: After observing one vehicle-function with its dependencies, the question arises how the architecture and interaction between multiple functions is set up. The viewpoint is not anymore one function, but observing multiple functions. The overall architecture of the software considering specific vehicle-functions should be visualized. Thus, third goal is to generate function-across architecture diagrams.

Practical Implementation

In the third iteration a vehicle-function-across architecture diagram, shown in figure 6.6, is created. A prerunning script analyzes dependencies based on "VehicleFunction1", "VehicleFunction2" and "VehicleFunction3. Internal package dependencies do not interest in this case. This is why these three blocks at the bottom represent corresponding packages for vehicle-functions. Package across dependencies should be observed. So, other blocks represent external software modules e.g. measuring technology modules, runtime environment and configurations. Obviously many software modules inherit to both vehicles-functions. Later the RTE is analyzed. This is why RTE-files are named separately.

Observing the dependencies, it can be determined that vehicle-functions are not directly in relation to each other. Indirect relations via external software modules can be traced (e.g. VehicleFunction1 > module2 > VehicleFunction3).

Evaluation

Goal was to visualize vehicle-function across architecture diagrams by viewing on multiple vehicle-functions and detect dependencies between. With the aid of a script external software module dependencies could be successfully detected. Architecture, hierarchy and inheritance are visualized for multiple vehicle-functions.

The second goal, detecting relations between vehicle-functions, could not be achieved. No diagram shows direct relations between vehicle-functions. As already mentioned indirect dependencies can be manually traced. However this does not mean that both functions depend on each other. An indirect dependency (e.g. VehicleFunction1 > module2 > VehicleFunction3 in 6.6) only signals: "There can be a signal flow between two vehicle-functions".

Consequently a static source code analysis, done by the batch-script, is not sufficient to detect direct dependencies. This leads to further developing.

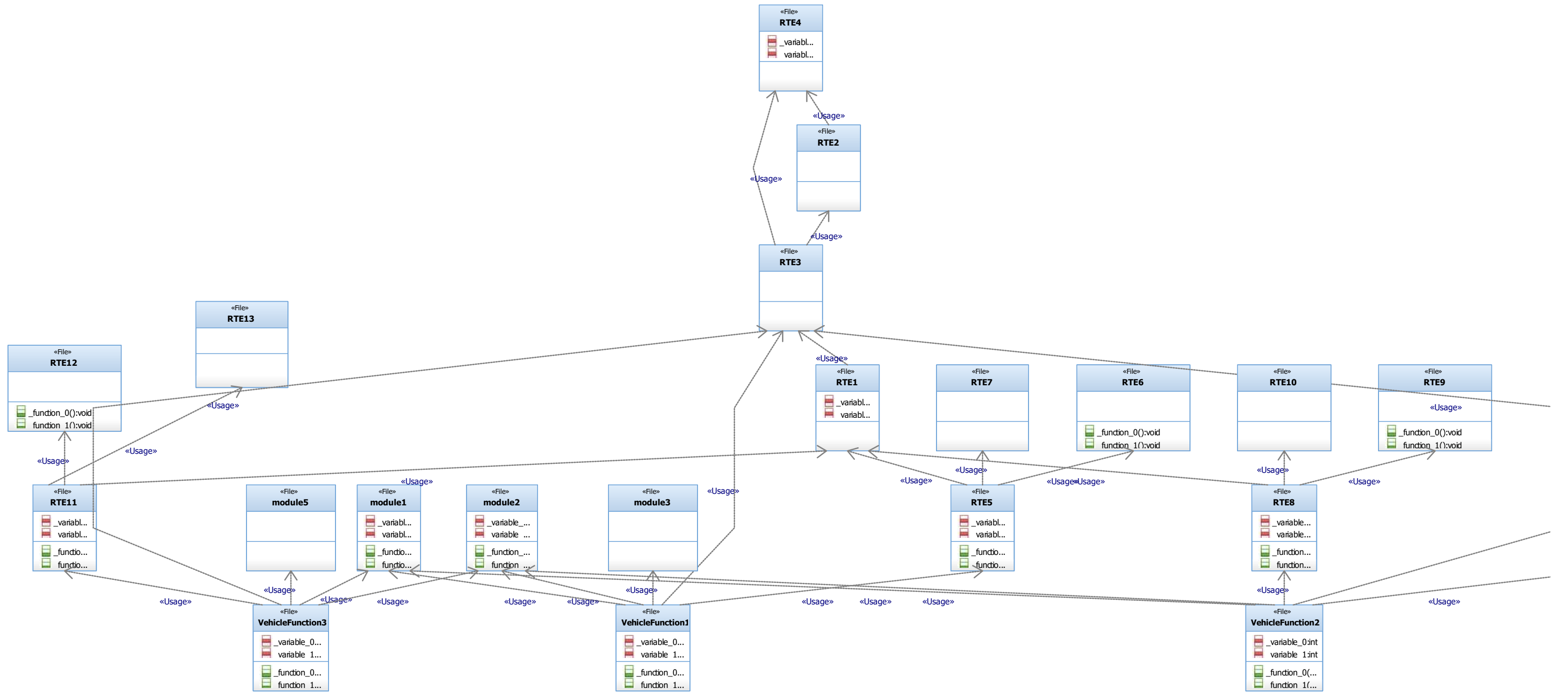


Figure 6.6.: Diagram for Vehicle-Function-across Architecture (data obfuscated), own representation with Rhapsody [Cor08]

Development of the Integration Model

First, the second script for preparation function across architecture diagram was included to Rhapsody similar to the first script as tool-functionality.

After that the runtime environment (RTE) was analyzed. The runtime environment is responsible for information exchange and communication between software modules 2.3.2. With manually tracing signals on the ECU, read and write accesses could be detected. This shows that several signals flow from a vehicle-function write access over the RTE to another vehicle-function as read access. Consequently signals with a write-read access from two vehicle-functions depend on each other. Furthermore, write-write accesses are avoided in software development. Read-read accesses will not interest in this thesis because they would not imply a direct dependency.

To automate the detection of direct dependencies i.e. read-write accesses, a Java API was implemented which analyzes signal flows of multiple vehicle-functions. Finally, it visualizes all relations between selectable vehicle-functions in a new architecture diagram. Additionally the trigger signal for the dependency is saved too. This Java API realizes the third functionality of the integration model, implemented in Java and added into Rhapsody via "Tools > Automotive Reverse Engineering > Dynamic Dependency Architecture Diagram".

6.3.4. Fourth Iteration

Objective

Generation of function-across architecture diagrams with signal flows: Detecting static dependencies lead to the question if signal flow dependencies can be analyzed. These dependencies should be detected by analyzing signal accesses of vehicle functions and visualized automated in a architecture diagram.

Practical Implementation

Diagram 6.7 shows a generated diagram with the Java API. Source code of the vehicle-functions has to be already imported. After running the Java API and analyzing signal flows of the RTE, for each read-write access a dependency is drawn. In this case the relation points from the reader to the writer. In other words: the reader depends on the writer of the signal. The specific signal name can be read off by clicking into dependencies. Via properties the signal name is saved.

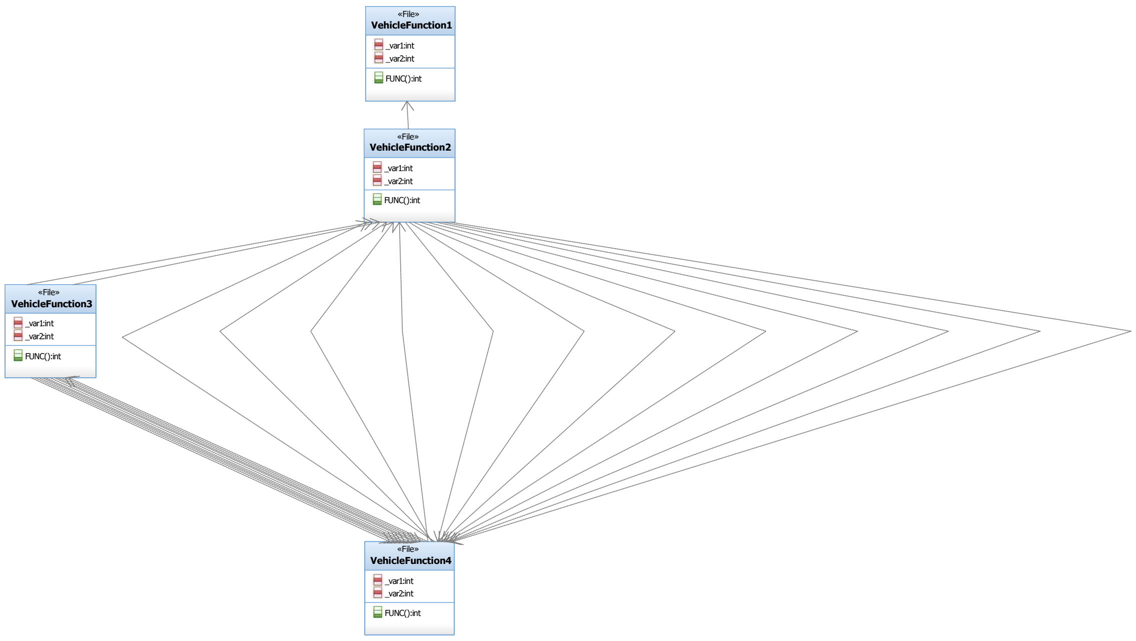


Figure 6.7.: Signal Flow Diagram (data obfuscated), own representation with Rhapsody [Cor08]

Each association represents a dependency. The diagram still has a clear representation because the maximum of dependencies between two vehicle-functions are already tested (twelve), which can be clearly visualized. Furthermore, the most common use case is to observe two vehicle functions. So diagrams will have a good clarity.

Evaluation

The goal to detect direct dependencies with signal flow analysis could be achieved successfully. The result can be visualized clearly in an architecture diagram, where each block represents one vehicle-function. This makes the architecture clearly recognizable. It should be noted that this functionality is adapted for the used software. More specific, the Java API is only usable for Matlab/Simulink auto generated source code. Yet, that forms the largest part of the ECUs software.

Finally the usability seems to be improvable. In further development software engineers should be able to apply the integration model intuitively. The function of generating signal flow architecture diagrams will be applied by only a few developer. Thus, the user interface was prioritized lower.

Development of the Integration Model

To finalize the integration model and its functionalities, an additional script for preparing signal flow dependencies is written. This script is callable via "Tools > Automotive Reverse Engineering > Preparation Dynamic Dependency" and restructures the software by only importing the main files of vehicle-functions. Diagram 6.3 shows that the main software module inherit all variables to every internal software module. After computing in internal software modules, the signals are given back to master RTE over main-files. This is why a main file is sufficient as representative for one vehicle-function.

Furthermore, a Rhapsody Profile is created to summarize all functionalities of the integration model and make them transferable to other user. This profile includes three functionalities to prepare reverse engineering and one Java API for dynamic dependencies.

Chapter 7

Prototyping

This chapter selects a suitable example of the productive environment and applies the integration model. The results form the basis for final evaluation.

7.1. Selection of an Example

For prototyping a suitable source code example needs to be selected. The integration model was developed for the productive environment. That is why the proceeding will be applied practically oriented. Therefore, different criteria need to be fulfilled:

- Software from ECU Intelligent Drive Controller (FAP)
- Current developing source code example
- Contact to software developer for appropriate evaluation
- Matlab/Simulink auto-generated source code

For developing the integration model in chapter 6, the whole software of ECU for FAP is used. Now, three specific vehicle functions named "AssistFunction1", "AssistFunction2" and "AssistFunction3" are chosen from the driver assistance package. The source code of these functions is auto generated Matlab/Simulink. All are actually in development for the upcoming "Fahrerassistenzpaket". This is why all data are obfuscated again.

Finally software developers evaluate the results. Thus, "AssistFunction1", "AssistFunction2" and "AssistFunction3" are well suited for a meaningful evaluation and examples for prototyping.

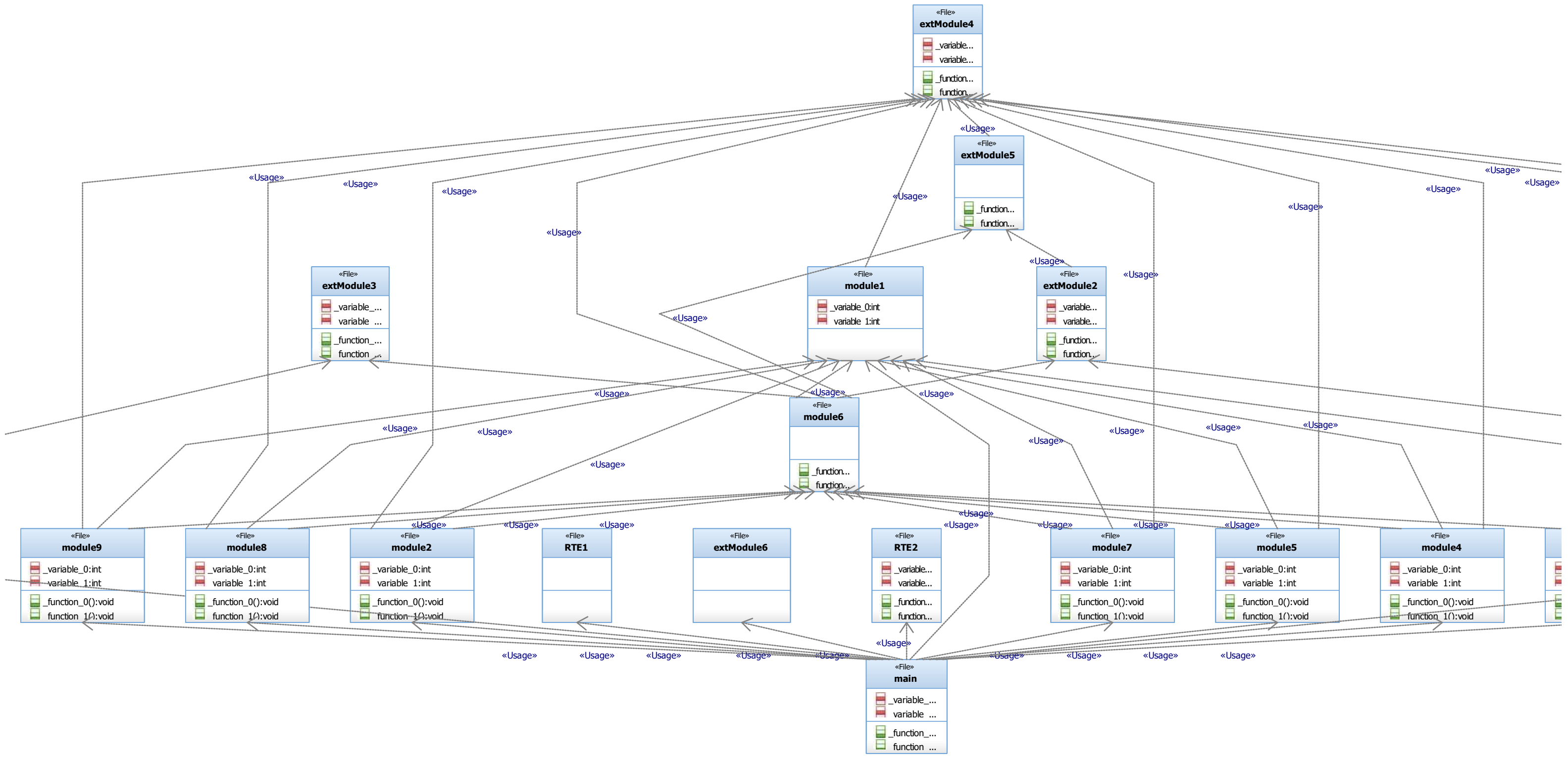


Figure 7.1.: "AssistFunction1" Specific Architecture Diagram, own representation with Rhapsody [Cor08]

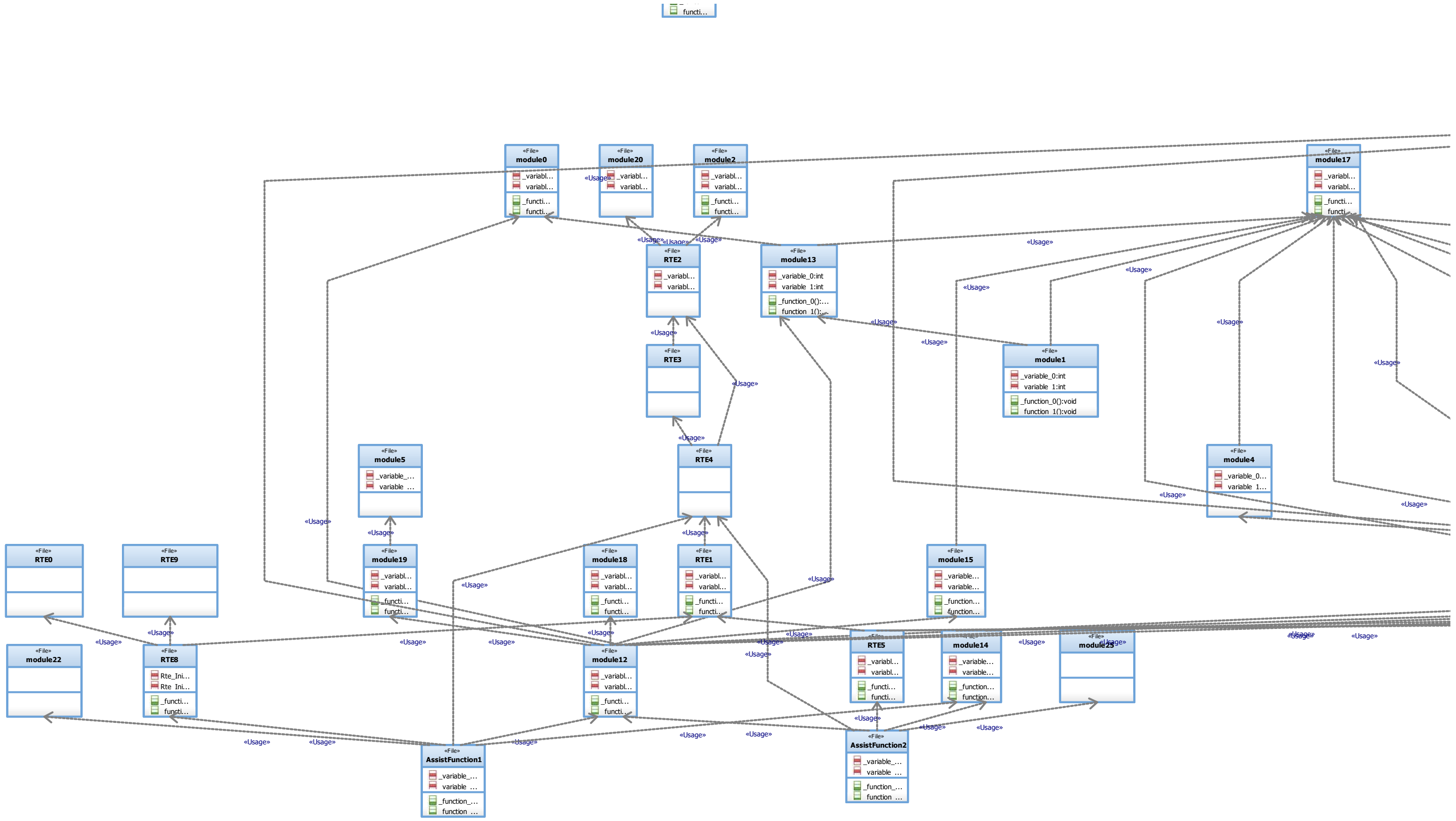


Figure 7.2.: "AssistFunction1 - AssistFunction2" across Architecture Diagram, own representation with Rhapsody [Cor08]

7.2. Practical Implementation

In this section all functions of the integration model are applied on the selected vehicle-functions "AssistFunction1", "AssistFunction2" and "AssistFunction3". So three architecture diagrams are generated with reverse engineering in Rhapsody with the aid of prerunnung scripts and JAVA API. A detailed description for the integration model is written for the internal usage and part of the appendix.

7.2.1. Vehicle-function specific Architecture Diagram

First, a vehicle-function architecture diagram is generated. After preparation script runs, the source code is imported by reverse engineering to Rhapsody. Obviously the architecture is very similar to diagram 6.5 because Matlab/Simulink uses automated mechanism for code generation.

7.2.2. Vehicle-function across Architecture Diagram

Secondly a vehicle-function across functionality is applied. The preparation runs for restructuring and after that, both functions were imported into Rhapsody with reverse engineering. Diagram 7.2 shows the architecture of AssistFunction 1-3. As already analyzed in 6.6, both functions are depending indirectly on each other.

7.2.3. Signal-flow Architecture Diagram

In the last instance a dynamic dependency architecture diagram, analyzed by static signal flows, is generated. First, a preparing script restructures the software modules. In the next step these modules are imported via reverse engineering. Lastly the respective JAVA API is running for analyzing read-write accesses for AssistFunction 1-3. In practice software developer mostly want to observe two vehicle-functions. This is why the diagram looks very simple. AssistFunction2 depends on AssistFunction1 because AssistFunction2 reads Variable2 while AssistFunction1 has write access. Simultaneous AssistFunction2 and AssistFunction3 have several read-write accesses to each other.

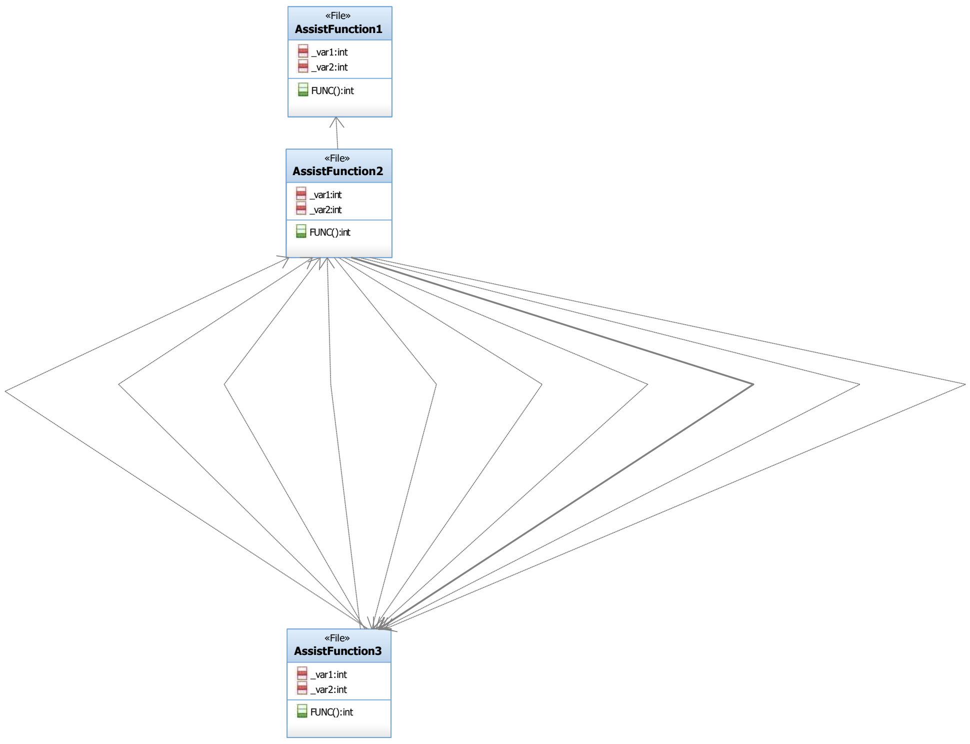


Figure 7.3.: "AssistFunction1 - AssistFunction2 - AssistFunction3" signal flow Diagram, own representation with Rhapsody [Cor08]

7.3. Result

As a result the integration model was realized as Rhapsody profile. Users can generate three architecture diagrams for different use cases. These models are located in three packages for each functionality. Based on this prototype the final evaluation in 8 is done.

This profile takes part of the project SEED product portfolio. This means that each developer with access to Rational Rhapsody can import this profile. Inclusion into the product portfolio was realized by a sprint process. The integration model was checked with external tool specialists for Rational Rhapsody. After that, several steps were passed through to become part of the product portfolio:

1. Inclusion into the sprint process
2. Definition of test cases
3. Testing the Functionalities

4. Detailed Presentation with Stakeholders
5. Sprint Review with short Presentation
6. If test cases passed: Inclusion to the product portfolio

After passing this process the integration model is included in the productive environment and downloadable for all developers at Daimler AG.

Chapter 8

Evaluation

This chapter introduces the final evaluation and takes review about the research questions of 1.3. First the research questions are listed. Then objective and subjective evaluation are executed. At the end, the results are presented.

8.1. Research Questions

Goal of evaluation is to give a final review of the results of prototyping. RQ1 is not scope of the evaluation. RQ1 is answered in chapter 2 by presenting a software developing process. This process is common in the automotive industry and similar to the software development process of ECU for FAP 5. RQ2, RQ3 and RQ4 are evaluated based on the objective 8.2 and subjective evaluation 8.3:

- RQ1: How does the software development process work in the automotive industry (at Daimler AG)?
- RQ2: Do software developer yield advantages from generated model with reverse engineering?
- RQ3: Which advantages can be yield using reverse engineering?
- RQ4: Can additional information can be visualized with reverse engineering in Rhapsody?

To unify the scale of both, evaluation are executed by rate the criteria from one to five. One represents the best and five the worst rating. Goal of objective evaluation is to rate the quality of UML architecture diagrams. The advantage for software developer can not be measured by objective criteria. That is why software experts took part in a

Table 8.1.: Objective Evaluation Criteria

Cluster	Criteria			
Perc. Organization	Associations	Crossings	Overlapping	Centering
Perc. Segregation	Symmetry	Orientation	Orthogonal arcs	Labels
Cogn. Effectiveness	Clarity	Distinction	Immediacy	Expressiveness

survey. These results rate the quality of models and benefits for the internal software development.

8.2. Objective Evaluation

The evaluation of generated models in this thesis is firstly done by chosen criteria from [MH08] and [SW05]. These criteria evaluate the quality of software models.

8.2.1. Criteria

All criteria are based on three different clusters: perceptual organization, perceptual segregation and cognitive effectiveness of UML architecture diagrams. All generated diagrams are evaluated based on these criteria from one to five (one=very good, two=good, three=satisfactory, four=sufficient, five=poor). The objective evaluation scheme is structured as following table 8.1:

Each cluster has several criteria to evaluate, which are explained below in short:

- Perceptual Organization: avoiding inheritance association, minimize crossings and bends, centering parents and children, positioning of superclasses, avoiding overlapping
- Perceptual Segregation: Symmetry of diagram, vertical and horizontal orientation, orthogonal arcs, horizontal labels
- Cognitive Effectiveness:

  Principle of semiotic clarity: redundancy, symbol overload and excess

  Principle of perceptual discriminability: visual distance, ease and accuracy

  Principle of perceptual immediacy: natural associations, logical similarities,

  Principle of visual expressiveness: number of different variables, visual notation

Table 8.2.: Objective Evaluation of Prototyping in 7

	Rating		
Cluster Criteria	diagram 1	diagram 2	diagram 3
Perceptual Organization	1.2	1.4	1.8
Perceptual Segregation	1.3	1.5	1.75
Cognitive Effectiveness	1.0	1.0	1.2

For the evaluation each criteria for every cluster is rated. For this thesis it is sufficient, only to represent one rating for each cluster. Therefore, for each cluster the average rating of criteria built. Table 8.2 represents the result of the objective evaluation. Here, *diagram 1* belongs to "AssistFunction1" specific architecture diagram 7.1, *diagram 2* belongs to "AssistFunction1 - AssistFunction2 - AssistFunction3" across architecture diagram 7.2 and *diagram 3* belongs to "AssistFunction1 - AssistFunction2 - AssistFunction3" signal flow diagram 7.3.

8.2.2. Analysis

The result of objective evaluation is mostly positive from 1 to 2 (very good to good). This means that the graphical representation is really clear and well structured. Especially diagram 3 is well rated. That is because figure 7.3 is a simple example with only three vehicle functions. Diagram 1 and diagram 2 are rated a little worse. It can be concluded that a diagram gets less clear if the number of model elements like association and blocks increases.

So in general, Rhapsody generates clear and well structured diagrams. The graphical representation can be adapted by visual modifications in Rhapsody. However the evaluation in context of visual representation is in the area of very good to good. Furthermore, this functions will not be used by a great number of users in productive environment. Just a few software experts will use these integration model. That is why visual modifications in Rhapsody have not to be adapted. Thus, generated models by the integration model in context of graphical representation are sufficient for the usage.

Table 8.3.: Survey probands

	Technical knowledge for
Proband 1	Rational Rhapsody & ECU software development
Proband 2	Rational Rhapsody
Proband 3	Automotive software development
Proband 4	Automotive software development (specific for FAP5)
Proband 5	Rational Rhapsody - Senior IT Specialist
Proband 6	Automotive software development research
Proband 7	Automotive Software development research
Proband 8	Automotive software development research

8.3. Subjective Evaluation

The subjective evaluation is realized by a survey with several internal and external experts for Rational Rhapsody and software development. First, the evaluation criteria respectively survey questions are introduced. After that, the results are analyzed.

8.3.1. Criteria

The goal of the survey is to measure the advantage and quality of generated diagrams. The questions of the survey aim to get a feedback about RQ2, RQ3 and RQ4. Furthermore, graphical representation is rated subjectively. Each parameter (in bold) is measured by a statement. Probands of the survey rate these statements from 1 to 5 (1 = agree, 2 = rather agree, 3 = neutral, 4 = rather disagree, 5 = disagree). The following statements were asked for each diagram of 7:

1. **Visual presentation:** The diagram is clear and well structured.
2. **Diagram content:** Information about dependencies and software modules are quick and easy to read.
3. **Source code readability:** Source code readability is improved in contrast from only source code to reading source code and diagram.
4. **Improvement on-boarding:** On-boarding processes would be easier with the aid of this diagram.
5. **Improvement software quality:** This diagram could improve the software quality.

The integration model will be used in the productive environment only by technical experts for software development or IBM jazz tools. Thus, just a few user will work with the integration model. That is why the survey was conducted with a few probands with expert knowledge for software development or IBM Rational Rhapsody. The participants are partially from internal at Daimler AG and partially from external service providers. All have expert knowledge for different areas. This is listed in table 8.3.

8.3.2. Analysis

Graphic 8.1 visualizes the results of the survey. Each dot represents the average of all probands rating from 1 to 5. At first glance it can be said that the result looks positive. The rating is in a range of 1 to 3 which means from agree to neutral. Goal of subjective evaluation is not to compare the selected diagrams from chapter 7. The generated diagrams should be a suitable usage for the productive environment. That is why the overall performance has to be considered for each criteria. Thus, each evaluation criteria is considered for all three models:

1. **Graphical Representation:** In contrast to the objective evaluation, the subjective is a little worse with regard to graphical visualization. But the rating is still positive with 2.0 to 2.5 (= rather agree). Diagram 3 contains the most amount of blocks and associations. So, presumably this causes a worse rating. Nevertheless all generated models have a clear representation of their architecture.

2. **Diagram content:** The next criteria has the same rating as graphical representation. Diagram 1 - 3 are rated from 2.0 to 2.5 again. This means that necessary information from models are mostly easy to read. Here, additional functions of Rhapsody are not considered. Users additionally have the opportunity to click into blocks or associations and get more information. By clicking into a block e.g. the user can get all information about the variables and functions. By clicking into a function, the user can get more specific information up to final implementation. levels.

3. **Source code readability:** The result of readability clearly shows that in contrast to only viewing the source code, the readability has been improved with generated diagrams. Obviously one diagram is better for getting an overview than only reading the source code.

4. **Improvement on-boarding:** Consequential to the improvement of readability, on-boarding new customers can be improved too. The rating is positive from 1.0 to 1.5. Obviously new employees who do not know the source code, can not see an architecture on the first view by just reading the source code. Generated

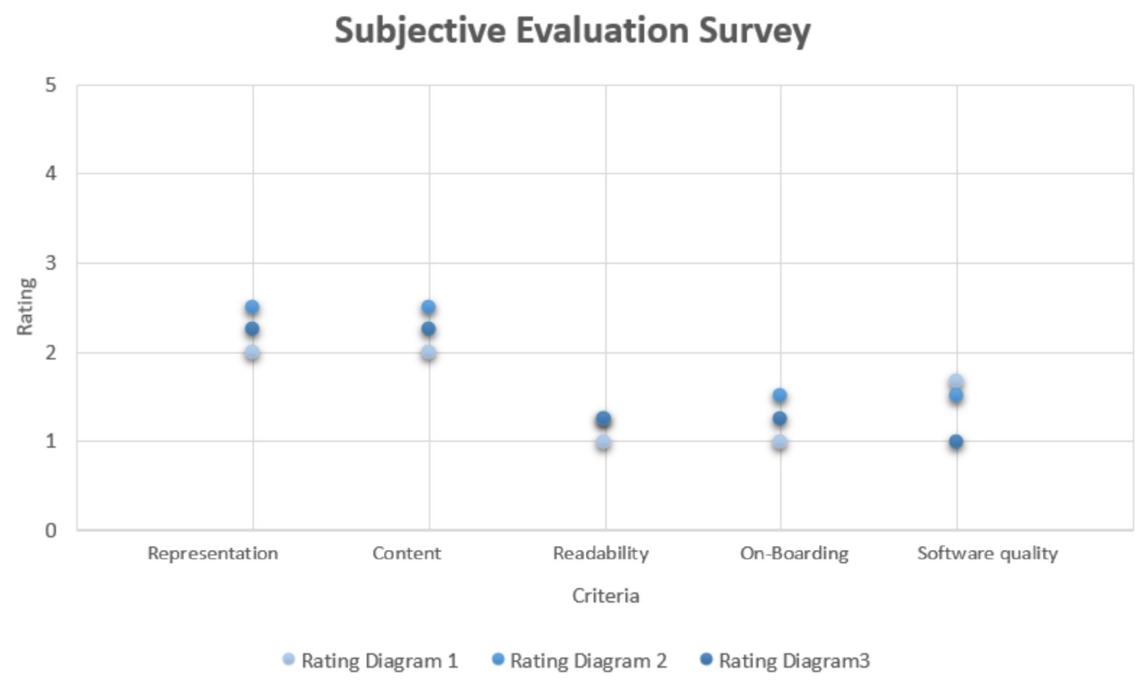


Figure 8.1.: Subjective Evaluation Survey

architecture diagrams support understanding the source code. Thus, on-boarding new employees is easier and faster with diagrams.

5. **Software quality:** Lastly, the software quality improvement is rated from 1.0 to 1.75. It can be recognized that diagram 3 is better rated than 1 and 2. This result was to be expected because diagram 3 contains a source code analysis of write and read accesses. Consequential this diagram contains information which only can be detected manually by analyzing signal flows. Hence, diagram 3 is a good approach to improve software quality. Therewith for example redundant dependencies or simultaneous accesses can be detected.

All in all the results of the survey are very good. It can be said that the main goal, to improve the readability and software quality, is achieved. The graphical representation of the diagrams has still potential. But all generated models are useful for the productive environment.

8.4. Result

First, the research questions of this thesis were reconsidered, then a subjective and objective evaluation were carried out. To answer the research questions and evaluate the quality of the developed integration model in this thesis, several criteria have been set up. The results of subjective evaluation as well as the results of objective evaluation are quite positive. According to ratings of software developers will yield advantages from generated models with reverse engineering (RQ2). These advantages include better on-boarding, improved readability and detection of dependencies (RQ3). Finally additional information could be visualized by detecting dynamic signal flow dependencies between vehicle functions (RQ4). Summarizing the result of evaluation is positive.

Chapter 9

Conclusion

Considering the increasing complexity of vehicle software systems, it is getting progressively more difficult to master this complexity. Abstractions of source code, like software diagrams, are necessary for large systems. Today's complex software systems, like automotive source code of ECUs, mostly miss a consistent architecture model. In practice, changes in development are implemented directly in the source code. Thus, corresponding models do not contain these changes. To generate consistent models, reverse engineering can be applied from source code to generate architecture diagrams.

Goal of this thesis was to develop an integration model with IBM Rational Rhapsody to generate UML architecture diagrams from source code. These diagrams should visualize software architecture of systems and yield advantages for the software development of Daimler AG.

The integration model was developed based on a ECUs source code. Therefore, source code, Runtime Environment and software structure were analyzed to improve the integration model. These modifications were implemented with scripts and JAVA APIs. Finally, the integration model is realized by a Rhapsody Profile. It will be used in the productive environment of Daimler AG. The Rhapsody Profile contains three functionalities: generation of vehicle-function specific architecture diagram, vehicle-across architecture diagram and signal flow dependency diagram.

It should be noted that these functions are adapted for one ECU. Even though source code is based on AUTOSAR, the integration model is not applicable for other ECUs without any modifications.

Future Work

This section introduces two approaches of the most important future works. There are many more future works for which this thesis can be used.

9.0.1. Transfer to other ECUs

Modern high-end cars contain more than 100 ECUs. Thus, the problem of missing architecture diagrams for source code is represented on each ECU. Even though source code is based on AUTOSAR, on deeper abstraction levels the architecture is built individually. This is why generation of architecture diagrams is important for each single software system i.e. ECU. Therefore, the transmissibility of this integration model should be tested. It is to be expected that this can be done with little effort. Furthermore, an automation for transmission can be considered.

9.0.2. Source Code Generation of Reverse Engineering Diagrams

This thesis explored about reverse engineering respectively proceeding against the direction of development. Thus, all generated diagrams are read-only and can act as documentation or development support. IBM Rational Rhapsody supports source code generation of C-Code too. Generated can be used as basis to generate automatically source code. It should be investigated if generated source code of reverse engineering models is practicable for software development.

Furthermore, Rhapsody provides the functionality to keep model and source code consistent. Changes in source code or diagrams are continuously synchronized with the whole software model. If this functionality is practically used in development, model-based software engineering can be realized more efficiently.

Appendix A

Use Case Description and Manual

Basic Reverse Engineering in Rhapsody

Projekt SEED Konzeption

Gedruckt von Kaiser, Shoma-Jakob (059)

28. November 2019, 11:15:14 CET

Konfiguration SEED Konzeption Master Stream

Konfigurationstyp Lokale Konfiguration

Komponente SEED Konzeption

Inhaltsverzeichnis

1 Basic Reverse Engineering in Rhapsody *3*

1.1 Introduction 3

1.2 Process cluster 3

1.3 Roles 3

1.4 Precondition 3

1.5 Postcondition 3

1.6 Scenarios 3

1.6.1 Scenario 1 - Usage of Reverse Engineering 3

1.6.2 Scenario 2 - Reverse Engineering for one software module 4

1.6.2 Scenario 3 - Reverse Engineering for the architecture of several software modules 4

1.6.2 Scenario 4 - Reverse Engineering for dynamic dependencies 4

1.7 Open points 5

1.8 Responsible Specification 5

1.9 Responsible Implementation 5

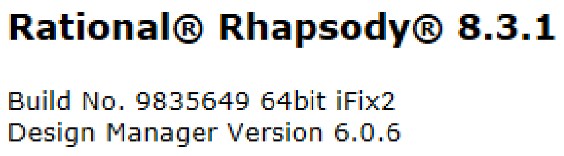
1.10 Stakeholder 5

1 <Basic Reverse Engineering in Rhapsody>

1.1 Introduction

This use case describes the usage of Reverse Engineering with IBM Rational Rhapsody in general and for a specific application for the FAP 5 source code.

Note(!): Used Rhapsody Version:



Rational® Rhapsody® 8.3.1

Build No. 9835649 64bit iFix2
Design Manager Version 6.0.6

1.2 Process cluster

1.3 Roles

Software Module Architect

1.4 Precondition

The source code has to be from the repository to a local drive. IBM Rational Rhapsody has to be installed. For Reverse Engineering of C-Code Rational Rhapsody Developer for C has to be used. The Rhapsody profile has to be loaded in.

1.5 Postcondition

Rhapsody will generate UML Architecture diagrams to visualize and detect the architecture and dependencies between several software modules. Furthermore, in a specific use case a Rhapsody Plugin will analyze dynamic dependencies between software modules.

1.6 Scenarios

1.6.1 Scenario 1 – Usage of Reverse Engineering

1.6.1.1 Step 1

Clone the source code in C to the local drive and start IBM Rational Rhapsody Developer for C.

A new Project has to be created where the diagrams are stored in.

1.6.1.2 Step 2

Switch to the "Advanced" developer mode (in the top left corner, right under the "save"-symbol.

With *Tools > Reverse Engineering* you can open the reverse engineering window.

1.6.1.3 Step 3

Choose the Code centric mode, select an input source as top folder and start the process by pressing "finish".

Rhapsody will load the entire project with the same folder structure into the project. On every package level there will be created object model diagrams which visualize the dependencies between the source code files.

Note: these diagrams just show dependencies in one package. These diagrams do not show dependencies package across.

1.6.2 Scenario 2 – Reverse Engineering for one software module

1.6.2.1 Step 1

This scenario is adapted for the dmc of FAP 5 source code. Clone the source code again locally. Press the button "*Tools > Automotive Reverse Engineering > Preparation Module Specific"*.

Enter the path of the FAP 5 directory and enter the module name (e.g. alc).

1.6.2.2 Step 2

A script is running to restructure relevant source code files. Run the basic reverse engineering (see Scenario 1) applied on the new created folder. *zz_bearbeitet > "Created folder"* . Choose the module folder and all files in there.

1.6.2.3 Step 3

This time one diagram is created for the chosen software module. All dependencies shown in the object model diagram are packing across. This means that this diagram takes the chosen module as point of view and visualizes all dependencies to other files in the whole software package.

1.6.3 Scenario 3 – Reverse Engineering for the architecture of several software modules

1.6.3.1 Step 1

This scenario is also adapted for the dmc of FAP 5 source code. Clone the source code again locally. Press the button ""*Tools > Automotive Reverse Engineering > Preparation Module Across"*.

Enter the path of the FAP 5 directory, number of modules for reverse engineering and enter the module names (e.g. alc).

1.6.3.2 Step 2

A script is running to restructure relevant source code files. Run the basic reverse engineering (see Scenario

1)applied on the new created folder. *zz_bearbeitet > "Created folder"* . Choose the module folder and all files in there.

1.6.3.3 Step 3

This time one diagram is created for the chosen software modules. This diagram shows the software architecture between several modules including the architecture of the Runtime Environment (RTE).

1.6.4 Scenario 4 – Reverse Engineering for dynamic dependencies

1.6.4.1 Step 1

This scenario is adapted for the dmc of FAP 5 source code based on Matlab/Simulink generated source code. Clone the source code again locally. Press the button "*Tools > Automotive Reverse Engineering > Preparation Dynamic Dependencies*".

Enter the path of the FAP 5 directory and enter the module names (e.g. alc).

1.6.4.2 Step 2

Right click on the project and call the function "*Automotive Reverse Engineering > Dynamic Dependency Diagram*". Enter the module names as one string (divided by spaces). After that enter the package name which includes the files. A new diagram is generated as "DependencyDiagram_*yyyy.mm.dd.hh.mm.ss*".

1.6.4.3 Step 3

The new generated diagram only shows the "main"-files. These main files represents one specific software module. Dependencies between software modules are based on a signal flow analyzation. A dependency is drawn if two software modules have a write/read-access if on the same signal.

1.7 Open points

Feature Document

1.8 Responsible Specification

-

1.9 Responsible Implementation

Shoma Kaiser (Daimler AG, ~~ITD/I)~~ supported external service providers

1.10 Stakeholder

Ansprechpartner aus ~~RD/A~~

Appendix A

Bibliography

[BLL06] L. C. Briand, Y. Labiche, J. Leduc. “Toward the reverse engineering of UML sequence diagrams for distributed Java software.” In: *IEEE Transactions on Software Engineering* 32.9 (2006), pp. 642–663 (cit. on p. 15).

[BM16] T. Briselat, S. Malewski. “Literaturrecherche.” In: (2016) (cit. on p. 21).

[BRJ99] G. Booch, J. Rumbaugh, I. Jacobson. “Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series).” In: *J. Database Manag.* 10 (Jan. 1999) (cit. on pp. 10, 11).

[Bun11] S. Bunzel. “Autosar–the standardized software architecture.” In: *Informatik-Spektrum* 34.1 (2011), pp. 79–83 (cit. on p. 12).

[CC90] E. J. Chikofsky, J. H. Cross. “Reverse engineering and design recovery: A taxonomy.” In: *IEEE software* 7.1 (1990), pp. 13–17 (cit. on p. 14).

[CN15] D. Cutting, J. Noppen. “An extensible benchmark and tooling for comparing reverse engineering approaches.” In: *International Journal on Advances in Software* 8.1&2 (2015), pp. 115–124 (cit. on pp. 19, 26, 27).

[Cor08] I. Corporation. *Telelogic Rhapsody User Guide*. Ed. by IBM. IBM, 2008 (cit. on pp. 16, 35, 36, 39, 42, 44, 48, 49, 51).

[FAK11] S. Freiberger, M. Albrecht, J. Käufl. “Reverse engineering technologies for remanufacturing of automotive systems communicating via CAN bus.” In: *Journal of Remanufacturing* 1.1 (2011), p. 6 (cit. on p. 20).

[FMB+09] S. Fürst, J. Mössinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkämper, G. Kinkelin, K. Nishikawa, K. Lange. “AUTOSAR–A Worldwide Standard is on the Road.” In: *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*. Vol. 62. 2009, p. 5 (cit. on p. 12).

[GHN09] H. Giese, S. Hildebrandt, S. Neumann. “Towards Integrating SysML and AUTOSAR Modeling via Bidirectional Model Synchronization.” In: *MBEES*. 2009, pp. 155–164.

[GHN10] H. Giese, S. Hildebrandt, S. Neumann. “Model synchronization at work: keeping SysML and AUTOSAR models consistent.” In: *Graph transformations and model-driven engineering*. Springer, 2010, pp. 555–579 (cit. on p. 23).

[GTD15] D. Gelles, H. Tabuchi, M. Dolan. “Complex car software becomes the weak spot under the hood.” In: *The New York Times online* (2015) (cit. on pp. iii, v, 7).

[Hee06] Heel. “Mercedes-Benz Automobile seit 1913.” In: *Mercedes-Benz Automobile seit 1913*. Motorbuch Verlag. 2006 (cit. on p. 10).

[HMS55] S. Henkler, J. Meyer, W. Schäfer. “Reverse Engineering mechatronischer Komponenten.” In: 5955.

[HMSN10] S. Henkler, J. Meyer, W. Schäfer, U. Nickel. “Reverse Engineering vernetzter automotiver Softwaresysteme.” In: *MBEES*. Citeseer. 2010, pp. 77–86 (cit. on p. 19).

[HVB+17] T. Huybrechts, Y. Vanommeslaeghe, D. Blontrock, G. Van Barel, P. Hellinckx. “Automatic reverse engineering of CAN bus data using machine learning techniques.” In: *International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*. Springer. 2017, pp. 751–761 (cit. on p. 20).

[KKM12] H. M. Kienle, J. Kraft, H. A. Müller. “Software reverse engineering in the domain of complex embedded systems.” In: *Reverse Engineering-Recent Advances and Applications*. IntechOpen, 2012 (cit. on pp. 15, 22, 23, 25).

[MH08] D. Moody, J. van Hillegersberg. “Evaluating the visual syntax of UML: An analysis of the cognitive effectiveness of the UML family of diagrams.” In: *International Conference on Software Language Engineering*. Springer. 2008, pp. 16–34 (cit. on pp. 28, 54).

[MRHK10] A. Michailidis, T. Ringler, B. Hedenetz, S. Kowalewski. “Virtuelle Integration modellbasierter Fahrzeugfunktionen unter AUTOSAR.” In: *ATZelektronik* 5.1 (2010), pp. 32–37 (cit. on p. 13).

[Ope05] C. of Operations. “Clarus.” In: 2005 (cit. on p. 8).

[PB06] M. Pagel, M. Brörkens. “Definition and generation of data exchange formats in AUTOSAR.” In: *European Conference on Model Driven Architecture-Foundations and Applications*. Springer. 2006, pp. 52–65 (cit. on p. 22).

[PBKS07] A. Pretschner, M. Broy, I. H. Kruger, T. Stauner. "Software engineering for automotive systems: A roadmap." In: *Future of Software Engineering (FOSE'07)*. IEEE. 2007, pp. 55–71 (cit. on pp. 8, 9).

[RG98] M. Richters, M. Gogolla. "On formalizing the UML object constraint language OCL." In: *International Conference on Conceptual Modeling*. Springer. 1998, pp. 449–464 (cit. on p. 11).

[Rup10] N. B. Ruparelia. "Software development lifecycle models." In: *ACM SIGSOFT Software Engineering Notes* 35.3 (2010), pp. 8–13 (cit. on p. 25).

[Sai14] A. Sailer. "Towards an automated reverse engineering of design models from trace recordings." In: *Informatik 2014* (2014) (cit. on p. 23).

[SBMH96] E. Sauerwein, F. Bailom, K. Matzler, H. H. Hinterhuber. "The Kano model: How to delight your customers." In: *International Working Seminar on Production Economics*. Vol. 1. 4. Innsbruck. 1996, pp. 313–327.

[SC08] M. E. sicherheitskritischer Systeme, A. von Codegeneratoren. "Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen." In: *Workshop auf der Modellierung*. Vol. 12. 14. 2008 (cit. on p. 17).

[SRH+11] S. Schmerler, T. Ringler, B. Hedenetz, U. Grüner, F. Wohlgemuth, C. Dziobek, P. Lohrmann. "Mit AUTOSAR zu einem integrierten und durchgängigen Entwicklungsprozess." In: *VDI (Hrsg.)* 15 (2011) (cit. on pp. 2, 7, 17).

[SW05] D. Sun, K. Wong. "On evaluating the layout of UML class diagrams for program comprehension." In: *13th International Workshop on Program Comprehension (IWPC'05)*. IEEE. 2005, pp. 317–326 (cit. on pp. 28, 54).

[Tea19a] A. W. Team. "AUTOSAR." In: *AUTOSAR-Homepage: https://www.autosar.org/*. 2019 (cit. on p. 23).

[Tea19b] A. W. Team. "AUTOSAR Introduction." In: *AUTOSAR Introduction*. 2019 (cit. on pp. 2, 12, 13).

All links were last followed on December 04, 2019.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Böblingen, 19.12.2019 S. Kaiser

place, date, signature