



HEAD OF DEPARTMENT

## BSc FINAL PROJECT TASK

**Zsolt Pásztor**

Mechatronics Engineering student

### ASPICE rendszerfejlesztési folyamat vizsgálata és alkalmazása

Az autóipar ismét abba az irányba halad, hogy egy vezérlő elektronikában egyre több és egyre bonyolultabb funkciókat kell megvalósítani. Ennek hatására a vezérlőegységgel szemben támasztott rendszerszintű követelmények száma és összetettsége is folyamatosan, gyorsan növekszik, mely új technikai és minőségügyi kihívások elé állítja az autóiparban dolgozó rendszermérnököket. Ezen kihívások egyike az ASPICE szabvány szerinti fejlesztés. Pásztori Zsolt szakdolgozatának feladata, hogy bemutassa az ASPICE szabvány által előírt fejlesztési folyamatot és annak alkalmazását a rendszerkövetelmény fejlesztés és a rendszer architektúra tervezés során.

Részletes feladatkiírás:

- Mutassa be az ASPICE fejlesztési folyamatot!
- Mutassa be a rendszerfejlesztés során alkalmazott fejlesztési eszközöket!
- Elemezze, hogy milyen kiegészítések szükségesek ahhoz, hogy ezek az eszközök kielégítsék az ASPICE szabvány követelményeit!
- Mutassa be az ASPICE szabvány szerinti rendszerfejlesztés lépéseit egy példa funkción!
- Valósítsa meg a szükséges kiegészítéseket és mutassa be, hogy ezek hogyan segítik a fejlesztési folyamatokat!
- Értékelje az elkészült munkát és adjon javaslatokat a lehetséges továbbfejlesztésekre!

**Tanszéki konzulens:** Dr. Hamar János, docens

**Külső konzulens:** Agócs Ferenc (Robert Bosch Kft.)

Budapest, 2015. december 7.

Dr. István Vajk  
Full Professor  
Tanszékvezető





HEAD OF DEPARTMENT

## BSc FINAL PROJECT TASK

**Zsolt Pásztor**

Mechatronics Engineering student

### Analyze and use the ASPICE system development process

According to the trends in the automotive industry the number and the complexity of the functions that should be implemented in one Electronic Control Unit are increasing. Due to this the number and the complexity of the system requirements are increasing very fast also, thus the system engineers are facing with new technical and quality insurance challenges. One of these challenges is the ASPICE standard for the development processes. In his thesis Zsolt Pásztor shall introduce the system development process according to the ASPICE standard and the use of this standard during the development of the system requirements and the system architecture.

Tasks in detail:

- Introduce the ASPICE Development Process!
- Introduce the tools that are used in Automotive System Development!
- Analyze how can be these tools extended to satisfy the requirements of the ASPICE Processes!
- Show the steps of the System Development in an example functionality.
- Implement the extensions and show the functionalities during the system development, how these extension help the system development
- Analyze the results and define further possibilities to improve the development process.

**Academic Supervisor:** Dr. Hamar János, assoc.prof.  
**Industrial Supervisor:** Agócs Ferenc (Robert Bosch Kft.)

Budapest, 16.09.2015.

Dr. István Vajk  
Full Professor  
Head of Department





Budapest University of Technology and Economics

Department of Automation and Applied Informatics

Zsolt Pásztori

Analyze and use the ASPICE system development process

BUDAPEST, 2015

Supervisor(s):

Ferenc Agócs

János Hamar, PhD

# Table of contents

|   |           |
|---|-----------|
| <b>Összefoglaló .....</b>                                       | <b>5</b>  |
| <b>Abstract.....</b>  | <b>6</b>  |
| <b>1 Introduction.....</b>                                      | <b>7</b>  |
| 1.1 Introduction of Aspice .....                                | 7         |
| 1.1.1 History and goal.....                                     | 7         |
| 1.1.2 Process dimension.....                                    | 9         |
| 1.1.3 System engineering process .....                          | 11        |
| 1.2 Tools used for development.....                             | 16        |
| 1.2.1 The developed product.....                                | 16        |
| 1.2.2 IBM Rational Doors.....                                   | 17        |
| 1.2.3 Sparx Systems Enterprise Architect .....                  | 19        |
| <b>2 Tool development for ASPICE processes .....</b>            | <b>22</b> |
| 2.1 Error checking and statistics generator tool (ECSG).....    | 22        |
| 2.1.1 Description of the task .....                             | 23        |
| 2.1.2 The implementation environment.....                       | 24        |
| 2.1.3 The run environment of the script.....                    | 26        |
| 2.1.4 The input parameters .....                                | 28        |
| 2.1.5 Statistics generation and regular expressions .....       | 29        |
| 2.1.6 Error checking with the help of I/O operations .....      | 32        |
| <b>3 System development according to ASPICE.....</b>            | <b>37</b> |
| 3.1 The functionality.....                                      | 37        |
| 3.2 Requirements engineering .....                              | 37        |
| 3.2.1 Analysis of the customer documentation.....               | 39        |
| 3.2.2 Requirement formulation.....                              | 40        |
| 3.2.3 Integrating the requirements into the documentation ..... | 43        |
| 3.2.4 Examples of requirement formulation .....                 | 46        |
| 3.3 System architectural design .....                           | 48        |
| 3.3.1 The SysML language .....                                  | 49        |
| 3.3.2 Package diagram .....                                     | 51        |
| 3.3.3 Block diagrams .....                                      | 52        |
| 3.3.4 Activity diagram .....                                    | 55        |

|                                  |           |
|----------------------------------|-----------|
| 3.3.5 State machine diagram..... | 57        |
| <b>4 Conclusion .....</b>        | <b>60</b> |
| <b>5 Table of figures.....</b>   | <b>61</b> |
| <b>6 Bibliography .....</b>      | <b>63</b> |
| <b>7 Supplement .....</b>        | <b>65</b> |

# HALLGATÓI NYILATKOZAT

Alulírott **Pásztori Zsolt**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2015. 12. 10.

.....  
Pásztori Zsolt

# Összefoglaló

Napjainkban egyre nagyobb szerepet kap az elektronika az autóiparban. Ahogy egyre több funkciót vesznek át az elektronikai komponensek, ezen vezérlők komplexebbé válnak, fejlesztésük nehezebb lesz. A komplexitás kezelésére jött létre többek közt a rendszer mérnöki feladatkör, mely rendszer szinten vizsgálja a komponenseket. Az autógyártók, hogy biztosítani tudják a termékeik minőségét létrehozták az ASPICE szabványt. Ez a szabvány a beszállítók fejlesztési folyamatát szabályozza és értékeli, kitérve a rendszermérnöki feladatokra is.

Dolgozatom során az ASPICE szerinti rendszerfejlesztést vizsgáltam meg és gyakoroltam magam is. Az elméleti ismeretek elsajátítását követően egy vezérlőegység fejlesztése során elvégeztem a követelményfejlesztés és a rendszer architektúra tervezés lépéseket egy kiválasztott funkcióra. A dolgozatban ismertetem az ezek során alkalmazott módszereket és eszközöket, továbbá szemléltető példákon keresztül bemutatom azok alkalmazását.

A fejlesztési folyamat során több alkalommal szembesültem vele, hogy egyes lépések automatizálhatók. A lépések automatizálásával a fejlesztési folyamat leegyszerűsíthető és felgyorsítható. Ezért a szakdolgozatom írása során több programot és scriptet írtam amelyek nagyrésze az IBM Rational Doors programot egészíti ki. A dolgozatom során ismertetem ezen programok fejlesztésének főbb lépéseit is.

# **Abstract**

Nowadays electronic components serve an ever greater role in the automotive industry. As more and more functions are handled by electronic components, these components become increasingly complex, their development at the same time becomes harder. To handle this complexity the system engineer role was born, this analyzes the components on a system level. The car manufacturers, to ensure the quality of their products, developed the ASPICE standard. This standard regulates and rates the development processes of the automotive electronics suppliers, including the system engineering role.

During my thesis I have analysed and exercised the system development according to the ASPICE standard. After acquiring the background knowledge I have prepared a component's requirement specification and architectural design. In my paper I describe the methods and tools used during the development, and give examples to them.

During the development process I have realized, that some steps can be automated. With the automation the development can be simplified and speeded up. Therefore I have written several scripts and programs amid the thesis work, most of them extend the capabilities of IBM Rational DOORS. In my thesis I have included the description of the development steps of these extensions.



# 1 Introduction

The more complex a system is the harder it is to design it in a way to be affordable while maintaining its reliability. Since the field of automotive electronics emerged in the last 30 years, it gave a big boost to the car industry, by helping the manufacturers design more comfortable and safer cars. In today's cars there are approximately 30-80 [1] [2] electronic control units, and an estimated 85% of all car functionalities are controlled by the electronics. The hardware and software manufacturing and development however differs greatly from that of a chassis, engine or transmissions, so big automotive manufacturing companies only designs the car electronics found in their vehicles on a system level, and orders the components from their suppliers, which specialize in the field of electronics.

The development of such component has to be fast, cheap and the original equipment manufacturer (OEM) has to provide full traceability during the process. In my thesis work I will refer to the electronics developer as supplier, and to the car manufacturers as customers. The ASPICE process standard contains the requirements and guidance to create a development process which fulfils the requirements of the automotive industry.

## 1.1 Introduction of Aspice

### 1.1.1 History and goal

Automotive SPICE [3] is a process standard, it is used for rating a supplier's development process in a zero-to-five scale, and it provides guidelines for the process development. It is based on the international standard ISO/IEC 15504 (SPICE), which is widely used for software development processes.

ASPICE is being developed by the Special Interest Group (SIG). The members of SIG include BMW Group, Fiat Auto S.p.A , Porsche AG and many more car manufacturers. Since 2007 these companies make their supplier assessment based on ASPICE. To become the supplier of these companies usually the suppliers have to reach a given assessment level according to ASPICE, so by implementing this model the OEMs can gain huge advantage over their peers.

|  |    |         |  |  |  |
|--|----|---------|--|--|--|
| Error correction costs today                             |    |         |  |  |  |
| Typical fault correction during:                         |    |         |  |  |  |
| Concept  | \$ | 1,300   |  |  |  |
| A sample   | \$ | 4,550   |  |  |  |
| B sample   | \$ | 5,200   |  |  |  |
| C sample   | \$ | 7,800   |  |  |  |
| PV series  | \$ | 84,500  |  |  |  |
| Production   | \$ | 104,000 |  |  |  |
| Post Production  | \$ | 117,000 |  |  |  |
| Source: HIS (Audi, BMW, Daimler, Porsche and Volkswagen) |    |         |  |  |  |

**1. Figure: Error correction costs [4]**

On Figure 1 the cost of a software error correction in a different stage of the development can be seen. Since currently most manufacturers can't update the car software remotely, even the smallest error threatening the passengers safety results in massive call backs, these are expensive for the manufacturer and also result in bad publicity, which can have a negative impact on sales.

ASPICE model helps the developers create a framework which is suitable for finding and correcting these errors in early development stages. The key point is the full traceability during the development process. Suppliers get the outline of the soon-to-be manufactured product from the car manufacturer in thousands of requirements. Each of these requirements must be implemented and tested, and from these steps documentation must be provided. This means that project members has to spend huge amount of time with writing documentations, but in the long run it has the advantage of reducing the chance of errors.

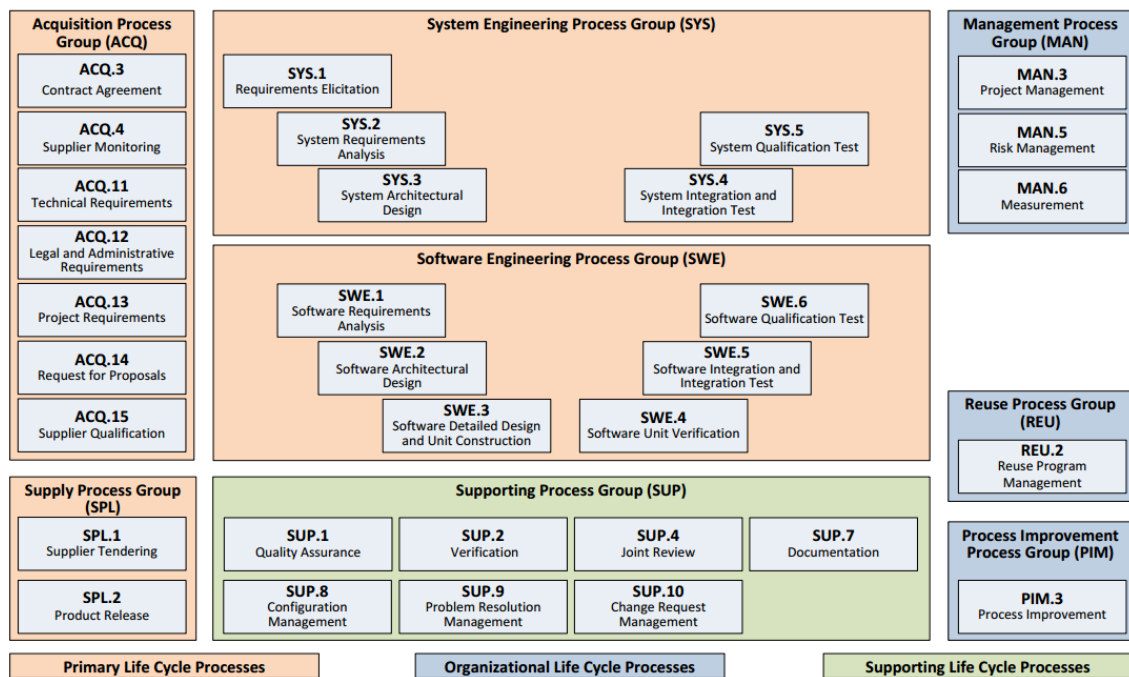
ASPICE takes over some of the most state-of-the-art notions. It doesn't only focuses on software, system and test processes during the development, but also on management, quality and suppliers. The idea is that to create quality the whole system must be of high quality. Another principle is that after creating a sustainable development process, the process development shouldn't stop, the processes have to be improved over time.

Automotive SPICE has two dimensions: process dimension which is based on ISO 12207 and process capability dimension based on ISO 15504. Based on these two features is the process maturity or ASPICE ranking determined.

### 1.1.2 Process dimension

The phrase process dimension in case of ASPICE means the internal structure of the process. It is the description of each developer group's (ex. system engineer, software architect) role during the process, and the connection between these actions.

The process dimensions are different for each project, even in the same company. They include not just the activities of the quasi developers, but also the support processes like management activities. Even though these activities can be different in case of separate projects the ASPICE manual gives an outline of the dimensions called ASPICE reference model.



2. Figure: Process dimensions according to the ASPICE Reference Mode [5]

In the primary life cycle processes, on Figure 2, we can see the system engineering process group (SYS) and software engineering process groups (SWE). These two groups can be ordered into a V shape, and they are mostly responsible for the product

development. The V shape is based on the so called V-model, or in German „Das V-Modell”.

The V-model is the graphical representation of the systems development lifecycle. It was originally created by the German Armed Forces in 1992. The main idea behind the model is, that the V letter has two sides, where the left side corresponds to the system and software development, and the right side to the test and integration of the software. Each process on the left side has a counterpart on the right, so all the development steps have a corresponding test, which guarantees that the design will be free from errors.

Another important feature of the development according to the V-model is that the different workgroup activities are not sequential. These means that in theory the software developers and software testers can work parallel, so the test cases are written before the actual function is finished. This makes development faster, but requires the communication of the developer parties, and the process steps might have to be repeated, if the work products between levels contradict each other.

It is not appropriate to say that the testing only takes place on the right side. During development we can differentiate between verification and validation (so called V&V). Validation corresponds to the question "Are you building the right thing?", This step must be taken during requirement formulation also, since the customer requirements might be erroneous. Verification corresponds to the question "Are you building it right?", and it enquires whether the product satisfies the customer requirements, so this is checked mostly on the right side of the V.

The creation of a concise process documentation has several advantages, which are the following:

- Minimizes the project risks, by improving the project transparency and control, which helps in the identification of deviations and risks during the project planning.
- Improves and guarantees the quality, this is achieved by the description of all activity outputs, which can be verified in early stages, and full traceability can be obtained.
- Helps in the reduction of costs, with higher transparency, and thanks to the clear activities during development no information and work product will be lost.
- Makes communication easier, both inside the project group and with suppliers and customers, by giving clear interfaces between the work groups.

During the planning of the process each step has to be named, it's purpose and outcomes must be specified. The process step outputs must be thoroughly described, so their performance can be measured. It is also important to describe each step in detail, to help the worker (so called stakeholder) fulfil it.

I will describe the system engineering process group (SYS) in detail, since that is in the main focus of my thesis.

### **1.1.3 System engineering process**

System engineering is a field of engineering which views the developed product in a holistic view. It is useful in case of complex systems, especially if the system has several different components, for example it has both hardware, software and mechanical parts. It focuses on the analysis of customer requirements, then proceeds with creating a system level design, while taking in account all the stakeholders and the project lifecycle.

The system engineering process steps are the following according to ASPICE reference model:

- requirements elicitation
- system requirements analysis
- system architectural design
- system integration and integration testing
- system qualification test

#### **1.1.3.1 Requirements elicitation**

This is the overall first step in a product development cycle. Requirement elicitation's purpose is to gather all available information concerning a product, and create communication channels between the customer and the supplier.

The system engineer must gather all information concerning the upcoming project. These information does not solely consists of the requirement documentation of the customer, it also contains the international and company standards which concern the product, the legal background information (ex. In case of pyrotechnics), and additional component specific documents, like hardware or mechanical documentation.

All of the relevant documents must be saved and baselined, always keeping in mind the consequences of a data loss, which can be dire, since these documents are the backbone of the project. In configuration management, a "baseline" is an agreed description of the attributes of a product, at a point in time, which serves as a basis for defining change. A "change" is a movement from this baseline state to a next state. The requirements which aren't available in written form, like decisions made during meetings, must be documented and added to the list of documents.

During this step of development proper communication channels must be created between the customer and the system engineer. This makes it possible for the customer to request changes in the requirements and track the development process. It also helps the system engineer to discuss directly with the consumer the requirements which can be vague, may contain contradictory statements or missing information. These problems can be solved by the consultation of the two parties. The consultations must also be logged and archived in a suitable framework, and the results can later be added to the main documentation as change requests.

#### **1.1.3.2 System requirements analysis**

The purpose of this step is to create the system requirements documentation from the customer documents. Sometimes the documents provided by the customer can be directly taken over, yet even in this case each requirement must be processed individually. However since Automotive SPICE gives strict guidelines against requirements usually completely new system documentation must be created, because of the high amount of work needed for this, system engineering plays a bigger role in ASPICE processes.

During the creation of the system requirements documentation the customer requirements should be used as main source. From these the system engineer must determine the functions and properties of the system. The functional requirements and additional information must be divided. In the meantime the requirements must be ordered into groups based on functional logics (e.g. Breaking system and motor control), relevant clusters for development (e.g. LIN messages and function logic), and development parties (e.g. hardware team and software team).

Each requirement must be thoroughly analyzed individually, and also in context of its dependencies, to determine its technical feasibility and its logical correctness. Also it is important to note the requirements technical impact and cost on the system. The

technical impact can be determined by looking at the function's interfaces and effect on the operational environment, without this for example a pulsing analogue signal output might be easily designed, but its voltage ripples can easily disrupt other functions, by looking at its environment we can however notice this hazard.

A requirement can only be used for development if its work product can be verified, either quantitatively or qualitatively. If a requirement cannot be verified, usually it means that it is not atomic enough, not well formulated or a necessary information is missing.

Automotive SPICE puts great emphasis on traceability. In case of requirement creation this means that each system requirement must have a source, which can for example be a customer requirement, a standard or some legal document. The requirement cannot be created from a discussion with the customer, unless it was documented earlier. This is the only way to ensure that the requirement can later be verified from customer requirement, and it also ensures consistency between the two documentations.

#### **1.1.3.3 System architectural design**

After the system requirement documentation has been prepared the system engineer has to design the system architecture. While the system documentation is mainly in a written form the architecture is a graphical representation of the complete system. The architecture is used as a base for software and hardware architectures.

The system architecture has two main component type, static and dynamic views. The static view shows the functional logics and the overall structure of the system. However the definition of the structure must be thought through, since usually there are different options, but each of these highly influence not just the system development, but how later it can be updated and what is the cost of the hardware and software changes. The evaluation criteria of the architecture depend on the project type, but this can be modularity, maintainability, security, scalability, completeness and even usability.

The dynamic view and the component interface definition helps the engineer see through the requirements and identify more complex problems. Some problems are hard to identify from only reading the requirements, for example the fact that a signal is missing from the function logics, or two functions are relying on each other in a way which makes them impossible to realize. However it is also important that while defining

the dynamic behaviour the system engineer shouldn't restrict the way later they can be realized by software or hardware engineers.

Even in case of the system architecture the bidirectional traceability must be maintained. This can sometimes be really hard, since the requirement documentation and a system model can be different in structure. While developing it the engineer must focus on consistency, and must communicate the main design choices with the consumer and other domains of the project, e.g. hardware and software development.

#### **1.1.3.4 System integration and integration testing**

In contrast to the last three activities system integration and testing is located on the right side of the V. Usually this process step is carried out by a different work group. The workgroups should be organized so the system integration and testing be as independent as possible from the system engineers. This makes the testing more precise, since the tester has to understand the requirements for the test cases, and the test cases will not be influenced by the implemented product, and how the implementing party's understanding of the requirements.

Usually the software and sometimes the hardware development are modular. In case of a complex system there can be more than a hundred software components. These individual components can only be integrated if the previous step, the system architectural design has specified the interfaces correctly. Integration even than can be troublesome, this is why usually this step is carried out by senior engineers with lots of experience in this field.

The system integration testing has to check the interfaces between the system elements, for example in case of an ECU it is done between the hardware and software. The first step of integration is the creation of an integration strategy, which specifies in which order the components are integrated. Before the integration, test cases must be created for each component and requirement. These test cases check the overall operation of the component in the system, but they focus mainly on checking whether the interfaces work between the components and whether they comply with the system architecture. These test cases must be periodically run on the whole system, and the test strategy must be regressional, so when a certain component changes in the system, the test should ideally remain valid. The integration testing of elements should include the testing of the availability of test signal flows between the elements, the timing dependencies of the



elements, the dynamic interaction between the connected items and the correct interpretation of the signals.

After the creation of the integration strategy and the test cases, the system integrators should stepwise connect the components. For each component the engineers should choose the valid system test cases and run them periodically. Test results should be logged and archived, to provide measurement values for the integration process.

While carrying out the integration and testing the V&V engineers must maintain bidirectional traceability. This means that the system architectural elements and system test cases must be connected, all item must have a connection in the system.

#### **1.1.3.5 System qualification test**

System qualification testing is the final step in the development process of a product. It does not just have to prove that the system is compliant with the customer requirements, but also that the product is ready for delivery. So this is the step during development which focuses most on the validation instead of the verification. In case of automotive electronics system qualification testing is done on an ECU level by the developer, but it is separately carried out by the car manufacturer on a car level too.

System qualification process steps are similar to the integration tests. First a qualification test strategy must be created, during planning regression strategies also has to be created, for easier handling of the product changes. Then the test cases must be specified and implemented. After their creation, the tests have to be selected according to the development schedule.

The most important step is the running of the selected test cases. The test results must be saved and archived, bidirectional traceability and consistency with the requirements must be maintained. Unlike during system integration testing where the tests can be run on a simulated system, or a so called „simbox” which is similar but not equivalent to the final product, during qualification testing tests are run on a real hardware, and in the last development state if the product is a component of a bigger system, it must be tasted „in the loop” too, to find out all possible errors, and to ensure its compatibility with all the other components.

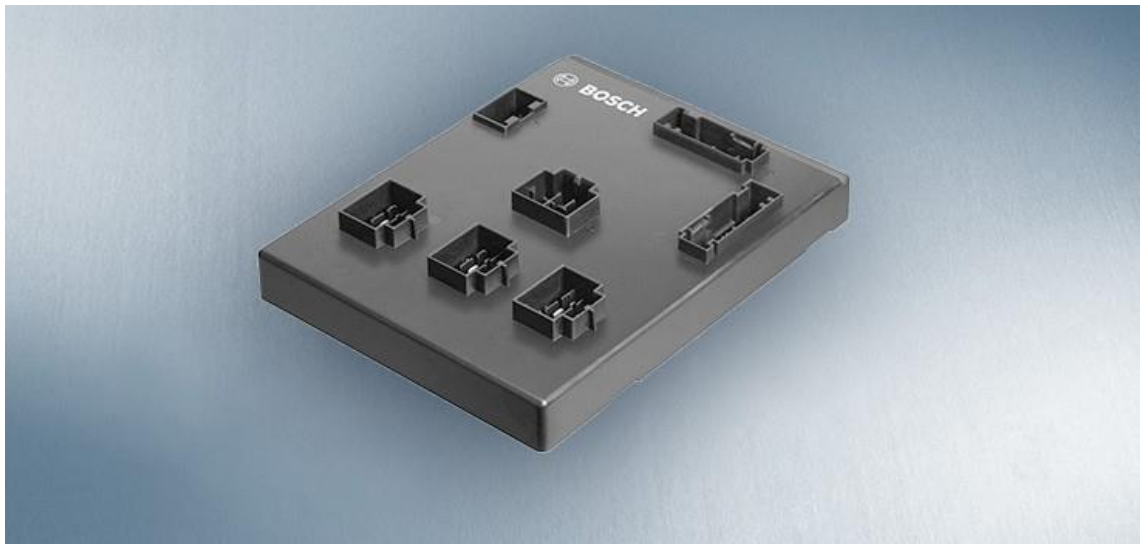
If the system qualification test results are positive, so there are no failed test cases, the product can be taken into production. This does not mean however the end of the

product development, since the car manufacturer can periodically order new features or change requests for the product, which will result in the rerun of the whole previous process.

## 1.2 Tools used for development

### 1.2.1 The developed product

The product, in whose development I have collaborated to during my thesis, is called the Bodycomputer module. An example can be seen on Figure 4. The Bodycomputer module (BCM) is an electrical control unit (ECU), which means that it includes a microcontroller and the required analogue and digital circuits, and memory. It is a computer module which is responsible mainly for controlling the car comfort functions, such as seat heating, inner and outer lighting and the wipers. It also works as a major communication hub in the automotive electronics system, hence has several CAN and LIN buses, and even Flexray and Ethernet connections. It acts as a server for distributed mechatronic components.



**3. Figure: Bosch Bodycomputer module [6]**

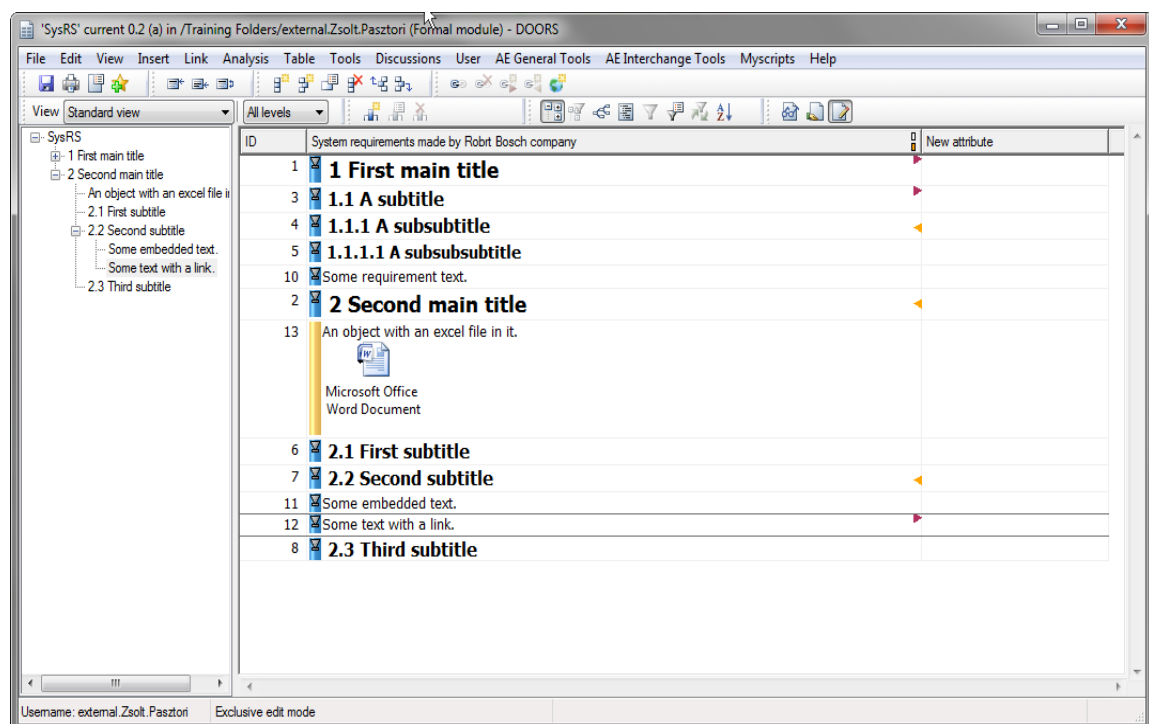
The BCM has four main domains: system, hardware, software and mechanic development. I personally worked on the system development aspect, however also programmed some for software tools. During the work process the main tools used by

system development are IBM Rational Doors and Sparx Systems Enterprise Architect. I will introduce these tools briefly in the next chapters.

## 1.2.2 IBM Rational Doors

Rational Doors [7] is a requirements management software developed by IBM Software Inc. . DOORS is an acronym, which stands for Dynamic Object Oriented Requirements System. It is used for creating and storing requirements in text form. The tool has many useful features for integrating it with other tools. It can be integrated with other IBM products, such as Rational Clearteam and Clearquest, and several file formats can be embedded into documents, supporting pdf, xls, picture formats and ppt. Rational Doors is widely used in the automobile industry by especially by manufacturers from Germany.

It is the main tool during system requirements analysis process step. The majority of the system requirements documentation is created during this step. However during all the development steps Doors is used, for a lesser extent, to log the changes and to link object for maintaining traceability.



4. Figure: The DOORS user interface

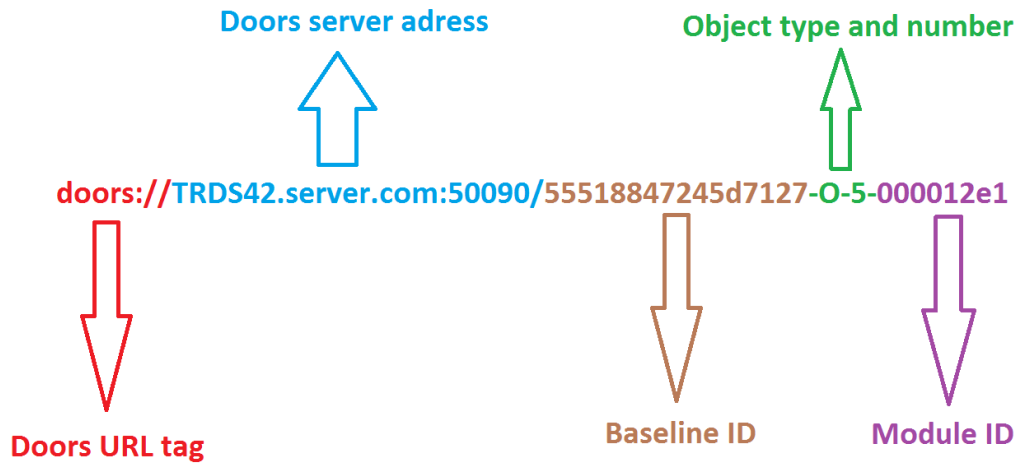
Rational Doors provides many advantages compared to traditional requirements management, which uses only archived folders, and store the information in pdf, word

and excel files. The most important feature is that the user can store all requirements ordered together. As the acronym of DOORS states, the program uses object orientated structure to store information. The highest level of hierarchy is the „Project” followed by „Folders”. Each folder can contain several „Modules”. Modules contain all the requirements, they can be viewed as separate files, a module can be compared to an excel file. On Figure 4 an example module can be seen.

The modules contain the „Requirements”, in an object oriented notation these could be viewed as child objects of the module. Requirements have headings and texts, but it is practical to use only one of them, to avoid confusion in the project. The requirements make up a hierarchical tree. Each requirement can act as a parent to later requirements. Requirements can have object properties, so called „attributes”. Some attributes are created automatically upon creating an object, for example „Creation date” and „Object text”, but others can be created by the user for additional data fields. Fields can be embedded into a requirement text field, Doors supports several file formats, for example pdf and excel files.

IBM Doors supports project work and archiving functions too. Doors is a client-server application. The files are stored on a server inside SQL tables, this makes it possible for several team members to access the data at the same time. The modules are downloaded to the client side when a user opens them, but only one user can edit them at once. When the user has finished editing the module he can save the changes, which will update the document on the server. To keep track of edits, Doors saves every change of every single item, which can be viewed in the item’s history page. The users can also create baselines of the modules, which can be later used as reference points during the development.

The software tool also has built in functions for traceability. First of all every single object created in Doors has a unique identification number. If an object gets deleted, its object ID never gets reused, this ensures that at no point will we find two equal object numbers. An object can be easily referenced from outside of doors, with the help of its unique legacy URL, whose structure can be seen on Figure 5:



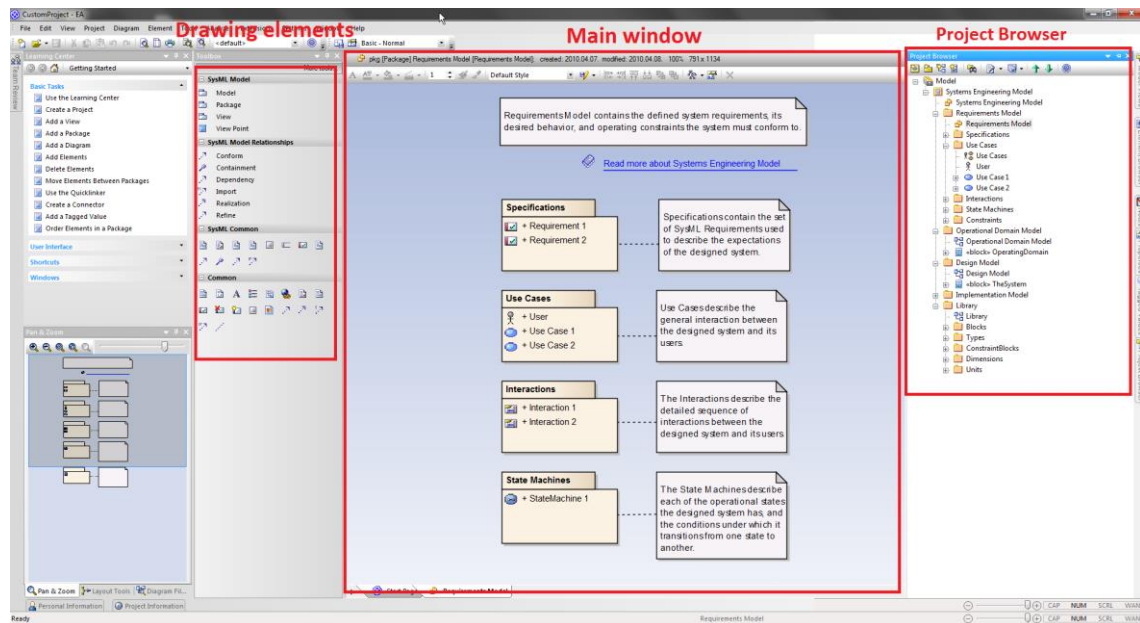
**5. Figure: The structure of DOORS legacy URLs**

On the other hand each requirement can be linked to other requirements. A link is denoted by a triangle. Incoming links have an orange hue, while outgoing links face the other direction and have red colour. As mentioned before outside files can also be embedded into objects, which means that traceability can fully be realized inside a DOORS project.

Rational Doors has its own scripting language, which is called DOORS eXtension Language, or short DXL [8]. DXL is based on conventional C language, with some traces of object oriented ideas, however it contains several restrictions compared to the original language. As a principle every function item inside Doors, and every action which can be made through the graphical interface can be automated by using DXL, even special GUI elements and server commands can be given using the language. In the later chapters of my thesis I will describe DXL language in more detail.

### **1.2.3 Sparx Systems Enterprise Architect**

Enterprise Architect (EA) [9] is a visual design and modelling tool. It can be viewed as an advanced drawing program, which is based on the Object Management Group visualization standards. Enterprise Architect is mainly used during the system architectural design. The system architecture is constructed with EA in the SysML language.

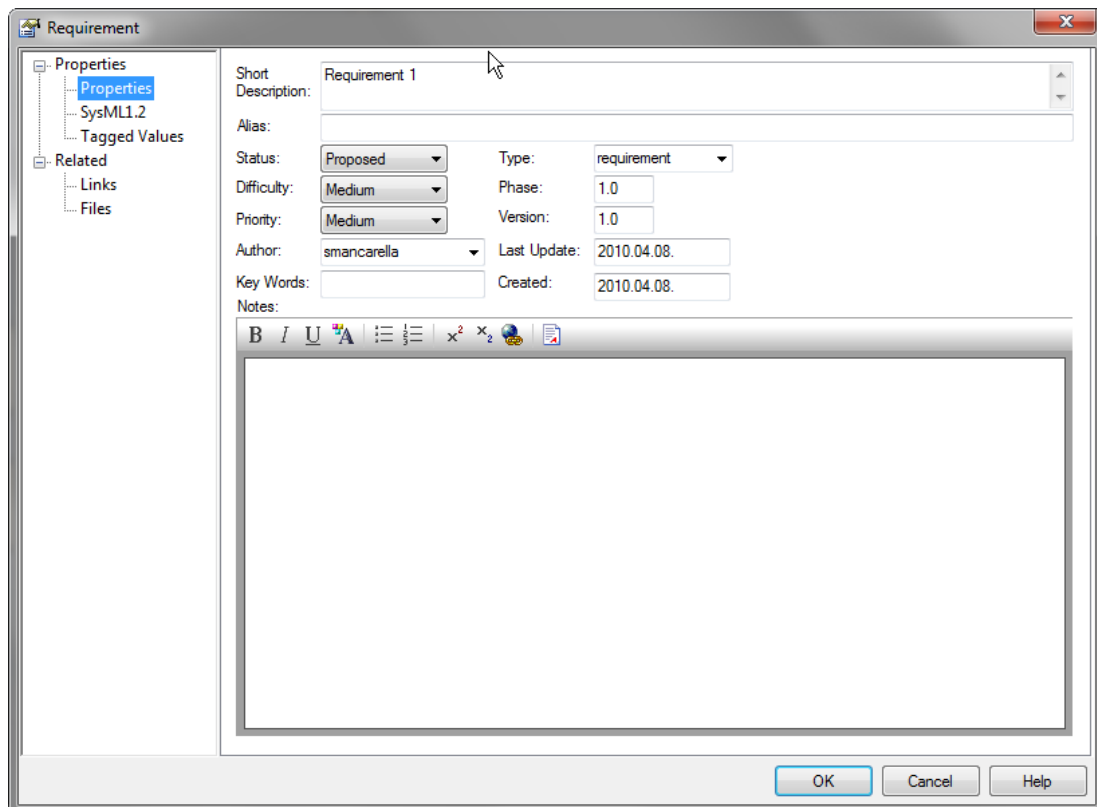


**6. Figure: The Enterprise Architect user interface**

Enterprise architect file structure is hierarchical. The top of the file tree is the project, project elements can be ordered into separate folders. The whole tree can be seen in the project browser, which is by default situated on the right side. The visual models are called diagrams, Enterprise Architect supports several standardized modelling languages, for example UML 2.5, SysML 1.3, BPMN 2.0 and ArchiMate. Different languages can use different diagrams, and diagram elements, so choosing the right language for the model is important.

Diagrams consist of elements and connectors. An element can be for example an activity, a decision node, or a user. Elements are child objects of the element class. When creating an element on a diagram, we can set the parameters of the element in the definition window, which is shown on Figure 7. An element can also be a separate diagram, so called composite element, which helps in describing layered systems. By double clicking a composite element in a new window its diagram opens.

To be able to depict information and material flows, the user can create links between elements. These links can have a specific exit point called port. Thanks to these ports the element interfaces can be easily visualized. Both ports and links have several properties, the user can specify for example their direction, the flow type (eg. information, item, control flow), the flow conditions and others.



**7. Figure: The element definition window**

Enterprise Architect can be run as a client only program, but it can also be extended for project use. In the latter case the files are stored on a Oracle Database, and a diagram can only be modified by one user at the time. For the tracking of changes in the database a different software called Rational Clearcase is used. This software can create baselines, which can be later used as archives.

The tool has additional features. It can be used for code generation in languages like C, C++, C#, Python and PHP. It also allows users to generate test cases based on the created code. Can also be used for data modelling for forward and reverse engineering, conceptual to physical level and reverse engineering of database schemas. It can be integrated with other tools, including Eclipse, CSV file handlers and Rational Doors. The integration with Doors allows the system engineers to realize bidirectional traceability between diagram components and requirements.

## **2 Tool development for ASPICE processes**

ASPICE standard requires the documentation of all development activities and also the full traceability between the work products that are made during the activities. Although following these guidelines results in a better quality software, it makes the development process more time consuming, since the developers instead of fully concentrating on their core activities, like programming or writing test cases, must spend time with writing great amount of documentation. During my thesis work I wrote tools and extensions which intended to help the developers by automating some of the documentation process. In the following chapter I will describe two scripts, one of them is responsible for error checking the written documentation the other is responsible for filling out specific information fields in DOORS documents.

### **2.1 Error checking and statistics generator tool (ECSG)**

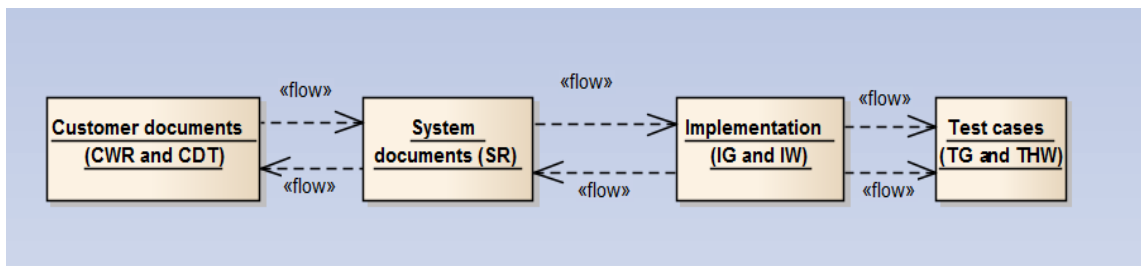
The project's DOORS database contains several tens of thousands of requirements. The customer can decide to regularly check the development process of the product, these occasions are called audits. During audits the auditors can randomly select objects from the Doors databases, and check their contents and dependencies, and if they find any error during their enquiries it might result in a bad audit grade, which can later translate into decreased item purchase from the car manufacturer. On the other hand if there is a problem with the product, without proper traceability it can be very hard to find the root cause, it can literally be like finding a needle in the hay stack, in this case the stack is thousands of requirements. Traceability and the rules of ASPICE creates an order in this stack of requirements which helps in finding the requirement which is responsible for the error, possibly sparing expensive work hours.

Because of the high amount of requirements, it is not feasible to check each requirement individually for compliance with the process. During the system development part of my thesis I realized, that luckily this activity can be automated, which makes it faster and the error checking can be repeated periodically. For this reason I have written a program, which goes through each element of all the valid modules and checks for errors, it also creates statistics to showcase the current standing of the database.



### 2.1.1 Description of the task

The Doors database of the project can be divided into levels, these levels are showcased on Figure 8. The top levels consist of the customer requirements, these can be separated into two sections, customer written requirements (CWR) and customer data tables (CDT). The next section is the system requirements (SR), which share the same template. The third section is the actual implementation of the components, which can be separated into implemented written (IW) and implemented tool generated (IG) items, the implementation can be in various platforms and formats, e.g. in case of hardware it can be a circuit diagram, for software components the C code. Finally the last section consists of the testcases, which belong to either the tests generated (TG) or the tests handwritten objects (THW). There are also other sections which I will refer to as other modules (OM).



8. Figure: The levels of the development documentation

The statistics generator script has 2 main uses. When run, it will generate 6 numbers which describe the standing of the SR, CWR and CDT modules. These numbers are:

- number of requirements in CWR and CDT
- number of linked requirements in CWR and CDT
- number of requirements linked to SR in CWR and CDT
- number of requirements in SR and CDT
- number of non-implemented requirements
- number of not-tested requirements

The numbers are added to a plot, which will make it easier to evaluate the current standing of the database.

The second feature of the script is a consistency check, which goes through all the valid SR modules, and examines whether a requirement is linked to an IG or IW object, its implementation state and whether it has a TG or THW object linked to it. These

information are saved for each system component separately, to help the developers in analyzing the database records which belong to them.

### **2.1.2 The implementation environment**

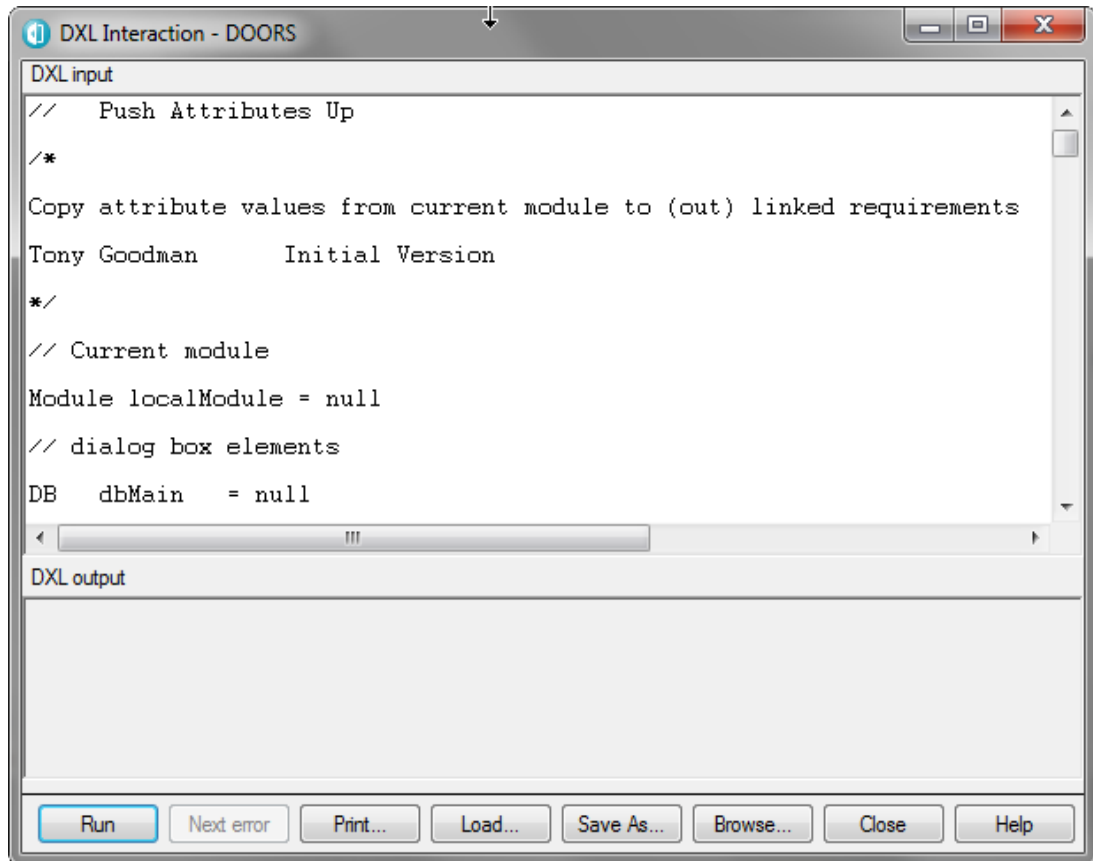
Although Rational Doors supports a rudimentary C API, I chose to implement the whole statistics script in DOORS eXtension Language (DXL), because it comes with a wide array of useful functions for handling the built-in data types of the DOORS environment.

The DXL language [8] is based on the C language, however it does not support the usage of references and pointers. It took over some of the features of C++. As the name of DOORS (Dynamic Object Oriented Requirements System) suggests the tool is heavily based on object oriented paradigms. The elements used in DOORS, like the previously mentioned, modules, attributes, objects etc. are all elements of classes. DXL can manipulate the elements of these classes, but in the language it is not possible to create new or modify already existing class definitions, it doesn't even support the C style structures.

Main language features:

- DXL is a statically typed language, using simple variable types, such as char, string, int, bool, void
- the fundamental types have garbage collection
- from fundamental types C style arrays can be created, e.g. `string[10] stringArray` is a string array named `stringArray`, which has 10 elements
- these arrays cannot be returned from functions
- for string parsing and editing regular expressions can be used
- in case of dynamic types mentioned before, which come from C++, memory must be directly freed with the `delete` command
- class elements can be referred to with the „current” tag, which is equivalent to the „this” tag in other languages
- functions can be created just like in C, but the function definition must always precede the usage of the function
- there are no function pointers
- since DXL is fundamentally a scripting language it has no predetermined main program
- other scripts can be embedded into a program by the `include` statement
- included files are inserted into the calling script, `include` works as if the programmer has copied the whole code of the included file into the script, the content is placed in the line of the `include` statement

- has easier syntax compared to traditional C, the use of semicolons and parenthesis are not mandatory, but not forbidden either
- exception handling in code is supported, however exceptions are returned only as strings, in code error handling should be avoided, conditions which can raise these errors, if possible, must be avoided



**9. Figure: The DXL interaction window**

DOORS scripts can be written and debugged inside the program environment. The DXL interaction window is the main tool for developing scripts in DXL, it can be seen in Figure 9. In the input field the user can type in the scripts, while the output box acts as a terminal for printing out outputs. By clicking on the „Run” button the scripts execution starts, if the script contains errors, the dxl output field will print out the corresponding error messages. For debugging the user can only use print messages embedded into the code, the stepwise debugging is completely missing. Unfortunately it lacks all the comfort features of state of the art development environments, and apart from the error message responses, the interaction windows does not provide more than a simple notepad. The tool used for writing longer DXL scripts is Notepad++ application, which has an extension for managing the DXL language, and has some useful features, like auto completion and colour coding for functions.

Since DXL scripts can only be used inside the DOORS environment, the language itself is not widespread, and has few users compared to other languages, such as Python or Scala. There is only one book written for DXL, which is the official reference manual. The book contains all the officially released functions, with description and examples, but it is not intended to be an educational material. Fortunately the official forum of the DXL language is a good source of information, and one can be sure to get a thorough answer to any question concerning DXL, thanks to the helpful and friendly community members.

### **2.1.3 The run environment of the script**

I have designed the script to be run on a server machine equipped with the Jenkins continuous integration tool. Jenkins CI [10] is widely used by software development projects, it provides a user interface for running automated processes, called jobs. These jobs are run after an event occurs, for example a developer uploads a new version of a code on the server, or the administrator can associate a timing schedule to the job.

Since several people edit the system documentation each day, the script must be run regularly to find the errors in the doors modules. By uploading it to a Jenkins server the administrator of the server can set the interval in which he wants the script to run. After each run the server logs the run parameters, for example the run time of the script, and whether it encountered errors, and archives the outputs of the script, which makes it possible to track the standing of the database. It is also considerable advantage of running the script on a server, that the runtime becomes shorter, because the server machine has higher performance, and better internet connection, than an average workstation.

**jenkins-debian-glue Continuous Integration labs**

| S | W | Name ↓                                       | Last Success      | Last Failure | Last Duration |
|---|---|--|-------------------|--------------|---------------|
|   |   | <a href="#">jenkins-debian-glue-binaries</a> | 8 min 6 sec (#1)  | N/A          | 2 min 49 sec  |
|   |   | <a href="#">jenkins-debian-glue-piuparts</a> | 5 min 12 sec (#1) | N/A          | 3 min 59 sec  |
|   |   | <a href="#">jenkins-debian-glue-source</a>   | 8 min 21 sec (#1) | N/A          | 9.9 sec       |

**Build Queue**  
No builds in the queue.

**Build Executor Status**

| # | Status |
|---|--------|
| 1 | Idle   |
| 2 | Idle   |

10. Figure: The Jenkins-CI user interface

While I wrote the script, the most important aspect was to write it in a way, that the code would be able to run completely automatically, which means that it should not require any user input during the run phase, and must be able to deal with run time errors and problems on the go. The run process of the code had to be logged, so the administrator would be able to find the error which halted the code execution. Finally all inputs and outputs had to be made available as output files, since information shown only on the GUI does not get saved.

The creation of the job for the script is done by the server administrator. The script runs in batch mode. During batch mode the user can't use the windows graphical interface and the standard inputs. The script can only write to the terminal window. Fortunately Rational Doors has a way to start script from batch mode.

The batch command is the following:

```
"c:\Program Files\IBM\Rational\DOORS\9.6\bin\doors.exe
-data 12345@address.de.server.com -batch C:\User\DxlScript\main.dxl
-osuser -a C:\User\DxlScript\"
```

The user must first specify the location of the Doors executable file. After the „-data” tag the server address and port number must be written. The „-batch” command tells Doors

that this will be a batch run and it is followed by the script file, basically our main program. The Doors folders can only be accessed with a valid username and password even in batch mode, „-osuser” will automatically provide the server username and password for the login information. Finally with the „-a” tag we can add a default folder, where we store the included files. There is a really useful however difficult to implement tag, it is called „-dxl”. This tag will let the user modify the soon to be run dxl script from the terminal by appending the string following the tag to the beginning of the script. This can allow outside programs to communicate with doors, but it is hard to implement it, since it is very difficult to debug it, it must be ensured that the additional lines don't break the code. I chose not to implement this tag in this project, because the script was too big and complex to test it in the first place, and this would have made it magnitudes harder.

After printing the batch command into the terminal the script begins its run.

#### **2.1.4 The input parameters**

The script could have been created as static code, without the use of any user input, however this would have limited its usefulness later. It would have meant that after any work process changes, if the system requirements are changed, the code would have had to be updated manually. Of course I could not allow this, since the code's complexity would mean that the slightest change in it would take hours to test. This can be avoided by loading in the main parameters from an additional file in the beginning of the script run, so the parameters can be easily changed.

The inputs come from a text file, with „.inc” extension. In the input file the administrator can set the script's run mode. He can choose from two options, the script can be run in debug mode or in normal mode. During normal mode, the administrator will get only essential information about the process, the terminal prints the start time of the session and the end of the session. While when run in a debug mode, the terminal gets flooded with runtime information. It will print out that at which step is the process, which helps identify a possible error, and it also outputs that which module is getting checked, and what are the results of the statistics for the module. In debug mode these information are also saved into a .csv file. This runtime log can help later identify possible errors.

It is important that when finding an error in the documentation the script is capable to identify the person who can correct the said error, so the error record can be directly forwarded, ensuring its fast correction. It shows the boons of traceability, that even a

script can reach a developer in case of an error, if the documentation is created according to ASPICE. The module names, and the responsible person can change frequently. The best way to identify the person responsible for the given Doors requirement is to maintain a document which connects the project members to software and system components. This document is stored in an excel spreadsheet, and gets updated regularly. The user must write the file path to this excel into the input file.

It is necessary to decide which modules are the valid modules in the documentation, since some of the modules can get outdated, deleted or renamed. The list of valid documents must be specified. Inside the modules there can be outdated elements or elements which has only been created, and are not yet finalized. The script is able to differentiate between the requirements based on their „age”, however the boundaries of the requirements validity must be regularly update and given as an input.

Finally the administrator can arrange that some modules will be evaluated differently than others. This can be given in a special syntax, which will be described in the next chapter.

### **2.1.5 Statistics generation and regular expressions**

The statistics generator part of the script is responsible for creating statistics which showcases the state of the database. This script opens the modules specified by the user and goes through each object in the module and checks conditions on them, then according to the result it may increment a counter.

First the script has to decide whether an object is a requirement or not, this step was probably the most difficult for me to implement, since it changes from module to module how requirements are identified. After finding a requirement it has to be decided whether it must be fully ASPICE compliment or only some of the ASPICE rules apply to it, this is the case with database table cells for example. If an object is requirement as the bidirectional traceability principle of ASPICE states it has to be linked to at least one another requirement on an upper and lower level of the documentation.

The script while examining the objects of the modules have to do string parsing, searching and identification, because the information in DOORS are in a written form. For working with strings there I could choose from two tools altogether. I could either use traditional string functions like find, replace and length determination. This kind of operations can be done inside DOORS with only one function called findPlainText, since

the function can be used for different actions, it has several input and output parameters, from which the script will not use most of, yet they must be created as variables. This kind of behaviour makes the usage of `findPlainText` uncomfortable, due to the fact that it leaves lots of clutter inside the code, and it is inefficient. Or on the other hand I could decide to use regular expressions.

Regular expression [11], or short regexp, are a sequence of characters that define search patterns, they can be used for pattern matching, string parsing, find and replace operations. They consist of metacharacters and regular characters. Metacharacters have special meaning, they can be logic operators for example. While a normal find string function can only find a complete match of the searched substring, a regular expression can be used to find several different variations of the given string. For example the regular expression „(A|a)nforderung” will return a positive match for the string „Anforderung” and „anforderung” but even for „Kundenanforderung”. In Table 1 I have listed the metacharacters currently available in DXL.

| Character | Meaning                                 | Example             | Matches  |
|-----------|---|---------------------|--|
| *         | zero or more occurrences                | a*                  | any number of a characters, or none                        |
| +         | one or more occurrences                 | x+                  | one or more x character                                    |
| .         | any single character<br>except new line | .*                  | any number of any character                                |
| \         | escape litteral text<br>character       | \.                  | literally a dot character                                  |
| ^         | start of the string                     | ^The.*              | any string starting with The                               |
| \$        | end of string                           | end\\.\$            | any string ending with end                                 |
| ()        | groupings                               | (ref) +<br>(bind) * | at least one ref string then any<br>number of bind strings |
| [ ]       | character range                         | [a-z]+[0-9]         | at least one small letter followed by<br>a digit           |
|           | alternative, logical or                 | (A a)nd             | either and or And  |

**1. table: Metacharacters currently available in DOORS DXL [8]**



With the help of these metacharacters I could create complex patterns. These patterns are stored as Regexp type variables in DXL. To apply a regexp variable one has to use it as a function, its only parameter is the string to be matched, and its return value is true if the pattern was found in the input string, or false otherwise. Thanks to this behaviour it can be used for searching for substring, and through patterns we can also eliminate the effect of mistyping, by including the most common mistaken characters into the regular expression. Regular expressions can also be used for extracting substrings. This can be done by creating groupings inside the variable with parenthesis, and applying the regexp to a string, after the regexp has returned with a value, we can extract the substrings corresponding to a grouping from the original string. For example we have a Regexp `re = „([a-z]*)([0-9].*)”` and the string `„agent007”`, we apply the regexp to the string, with `re(„agent007”) returning true`, we can type `re(match[2])`, which will return the second grouping of the match, which is `„007”`. This way we can easily parse whole files, by creating a difficult pattern a whole .xml file can get parsed in one step.

Recognizing whether a DOORS object is a requirement is a standard example for object oriented programming, since these kind of objects have the same structure, and the same meta functions should be run on them, by meta functions meaning abstract functions instantiated for each module differently. Unfortunately DXL does not support class creation, nor inheritance, the closest thing it has is C type arrays, coupled with the use of regular expressions. Since regular expressions can be viewed as functions with limited scope, but a uniform structure, we can use them for instantiating the necessary functions for the script. For this purpose I created a string format as an input, this was mentioned in the input section, and a more complex function to interpret these commands.

So the statistics script's inputs are string arrays. The user can specify as many input arrays as many different rule sets are for the system documentation. Each array contains four string elements, for example:

```
string MeasurmentValues[4] = {"00013th6","Object Heading,Object Classification,
Object Number","n/a,A|anforderung,([0-9]+)([a-z]?),","true,true,false,"}
```

The first item of the array is the identification number of the modules, which ruleset is valid to. The module number is used instead of the name to avoid confusion when a module is renamed. The remaining three items define the ruleset of the evaluation, each strings given elements belong together, for example `„Object Heading” <-> „n/a” <-> „true”` create a rule. The second array element specifies the function which attribute of

the object has to match, the third element specifies the matching pattern, while the third whether the match result should be negated or not. I needed to include this part because unfortunately in DXL the implementation of the negation operator in regexp (in Python „!“) is missing, which unfortunately limits this kind of usage of regexp, even though the „and” and „or” logic is present. Bottom line, with the help of the unorthodox usage of regular expressions, I have created a method for using classes in my DXL program, without DXL actually having these features.

The statistic scripts loop through the input arrays. It first opens the modules belonging to the ruleset, then goes through the objects one by one. Reads in the necessary attributes of the object, assigns default values to them in case they are empty, then matches the regular expression patterns to them. The return value of the matching is left untouched in case the expression has positive logic, or gets negated else. Finally the function decides if the object is requirement if all the final values are true.

After the function has selected the requirements it checks their outgoing and incoming links. If the links are connected to an element in the right layer of the documentation, for customer documents it is the system documentation, then the requirement follows the ASPICE rules, else it is missing information, and should be corrected.

## **2.1.6 Error checking with the help of I/O operations**

### **2.1.6.1 Reading Excel files from DXL**

The error checking part of the script was harder to implement compared to the statistics part. It not only needs to check more conditions for each item, but also has to be able to manage inputs given in .xls and has to output data in .csv format. The DXL language had no functions whatsoever for communicating with other programs, I had to try several approaches and file formats reading the inputs and writing the inputs. I chose the excel table and the .csv files because they are rather easy to manipulate, considering that I had to write all the functions for them.

As mentioned in the inputs section, the main input of this script is an excel data table, which contains the connection between project members and the components they are responsible for. The DXL language has no specific library and functions for managing excel documents, but it does support the usage of OLE objects, without this capability it

would be tricky to manipulate excel files, since their data structure is stored in a binary format.

The OLE acronym stands for Object Linking and EMBEDding [12]. It is a technology created by Microsoft, and it is mainly incorporated into Microsoft products, such as Microsoft Office. Its goal was to make it easier for windows compatible programs to communicate with each other, by giving them a standard interface. OLE interfaces can be utilized after creating OLE handlers, so called OLE objects. These OLE objects are the child objects of corresponding OLE classes in the client application, in this case Microsoft Excel. An OLE object can correspond to a whole excel document, a worksheet and cells. After creating a specific object, we can manipulate its contents with the help of OLE put functions, these are equivalent of the set methods in private classes. We can also retain their property values with get functions, and perform actions with OLE methods.

With the help of OLE commands I could access the data in the excel file by creating an OLE object for the file. Then a separate object had to be created for the sheets. Since a user can change the order of the excel sheets unintentionally, the program has to parse all the sheets and find the one with the important information. Then on the sheet with the help of an addition OLE object corresponding to the worksheet cells, the two columns containing the name of the project members and the belonging components had to be found. After these values are read from the sheet they had to be loaded into the script, and the Excel file can be closed.

During the coding of this function the biggest issue was not the accessing of the excel file, but opening it in a way that excel will not ask for any input from the user. Unfortunately Microsoft Excel has the habit of asking for user input after opening or closing a file in a form of a pop up window, this had to be restricted by setting several run parameters trough OLE, because a popup windows halt the whole script's execution.

#### **2.1.6.2 Jumping across modules with links**

After obtaining the information from the excel file the script can start the error checking. ASPICE process has strict rules for system requirements:

- system requirements must be linked to customer requirements
- system requirements must be linked to implementation documents
- implementation documents must be connected to test cases
- all implementation documents must be complete and accepted
- all test cases must be complete and accepted.

The technically problematic part is the checking of the traceability in DOORS. Traceability is ensured by the use of links. Links can be separated into outgoing and incoming links, unfortunately they have completely different properties if viewed from DXL. Each requirement has several links which have to be checked, opened and their target examined.

Modules contain all the information on their objects, the object attributes, the object history and the properties of the modules too. The amount of information stored in a module is reflected in their size, because of this it takes lots of time to load a module on client side. However after loading the information, which means downloading it from the server, they can be edited and read quickly, since dealing with strings is a quite efficient operation. Links are usually created between modules, if the script would open each module with the link then close them, the many module downloads would make it impossible to finish the action. While writing a program in DXL it is important to open every module only once to limit the amount of data to be downloaded, but at the same time the modules which are not needed anymore must be closed, to spare memory.

The data of the links is always stored in the link's source. So outgoing links can be examined from the parent module, and it is not needed to open the modules they are pointing at. On the other hand incoming links don't store information about the object they are pointing at. We can obtain their target, but only the target, from their link reference, which is a pointer. In order to access their target object their module must be first opened. To optimize the runtime of the script, it has to create dictionaries containing the name of open modules. When a link is examined, first the script looks it up in the dictionary, if it is unopened, the script opens it and appends its name and reference variable to the dictionary. If the dictionary already contains the module, the function only fetches its pointer. After the whole script has ended the items of the dictionary are closed one-by-one.

The error checking of the system requirements loops through all the elements of the valid modules. First the script determines whether an object is a requirement, and that it was created according to APSICE, this step is done with the use of regular expressions. Then if the object is a requirement it loops through the links connected to the object, checks whether it is linked with customer and implementation documents. If the requirement has the necessary links the script checks all the other objects connected to the requirements via links for their implementation status. Finally it opens all the implemented objects

connected to system requirements, and looks for their test cases, which should also be connected.

### **2.1.6.3 Outputs of the script**

For each object, which was identified as a requirement, the error checking steps must be noted down, and the requirement must be saved in a way, that they can be traced back to the DOORS objects.

After the script has read in the responsible developer and component pairs from the excel file, the output files are created. Each component gets its separate output file in .csv format. Hardware and software components are divided into separate folders. Two .csv files are created for items whose components are missing from DOORS and items whose components are not mentioned in the excel file (these are usually results of mistyping).

Comma-separated values files, or short csv, are plain text files, but they hold tabular data. Each line of the file is a record, lines are separated with EOL characters. The records can have several fields, separated by commas, more precisely semicolons. These files are frequently used for communication between programs, since they can be easily written as text files, and can be read by the developers. Csv files can be opened by any text editor, such as Notepad or Microsoft Word, however Excel can manage them as tabular data, and provides utilities for editing them.

I have written a function for DOORS to be able to write csv files. It is rather easy to write these files as text files, one only needs to watch out for line ending and semicolons which appear in the text to be saved, since the editors, can recognize these as line and field ends. These characters have to be exchanged to other whitespace characters, for example horizontal tabs or spaces. To handle a file inside DXL, a stream variable must be appointed to the file. With the stream variable DXL can write the file line-by-line. For the sake of optimization the script stores all the stream variables inside an array, and each time a record must be added to one of the files the csv function just reads out the stream variable from the array, this way it spares the time for opening the file, and finding the last line.

When the script has finished the error checking of a requirement, it chooses the output csv file, which stores components with the same name. To be able to trace back the requirement to its doors document the requirements module name, object number and

baseline version is saved. The results of the error check are also noted down, namely whether it linked to an implementation object, the state of the implementation object if it is available, and whether the requirement has an associated test case. Some additional data gets saved for debugging purposes.

## **3 System development according to ASPICE**

In the following chapter I will describe the system engineering process according to the ASPICE standard, with the help of a template function. The template function will contain the results of the following process steps:

- requirement elicitation
- system requirement analysis
- system architectural design

### **3.1 The functionality**

The functionality I have developed during my work is the terminal control function. The terminal control function is responsible for managing the battery voltage provided to the car electronics components. For example it provides voltage for the auto radio when the car engine has ignition. This function is very good example of the holistic development during system engineering, since it has both software and hardware aspects.

Terminal control is present in all cars regardless of manufacturer. The electric terminal numbers are standardized, they are included in the DIN 72552 [13] standard. The input values of the function come from several sensors, including the vehicle speed sensor, the door locking sensor and the ignition switch. On one hand the sensor supplies signals for internal functions, which describe the terminal state, and also provides battery voltage to appliances through relays, these appliances are for example the auto radio, outer lighting and the cigar-lighter.

In the current paper only the outline of the terminal control logics will be included, in a greatly simplified manner, because of the length and legal constraints.

### **3.2 Requirements engineering**

The requirement engineering step of my thesis work contains both requirements elicitation and system requirements analysis steps according to ASPICE. During this step I have created the system documentation of the terminal control function.

The system documentation is stored in the DOORS tool mainly in text form. The most important items in the documentation are the requirements, however there are other items too:

- heading: Headings separate the documentation into sub-categories. They improve the understandability of the document, by ordering the items into associated groups. Headings however don't contain information concerning the function implementation, and they cannot be tested.
- information: The requirements sometimes rely on background information, or the purpose and use case of the requirement can help the understanding. These pieces should not be added to the requirement body, but can be stored in the document as information objects. These objects don't need test cases, and the traceability doesn't have to be strictly maintained.
- constraint: These objects contain guidelines for the development of the product. They contain specific solutions, specified by the customer.
- requirements: They are objects specifying a certain attribute, characteristic or quality of the product. Requirements must have associated test cases, and their traceability must be maintained.

The first step of the creation of the documentation was to collect all the data, which had any connection to the function's development. The main source of the work was the requirements list given by the customer. The easiest way to derive the system documentation from the customer documentation would be to simply copy the items, take them over directly. However, no matter how tempting this approach is, it is unacceptable in case of ASPICE, since every single customer item must be processed with requirement analysis. Customer requirements and additional information must be separated. As a rule of thumb a requirements tells something that the supplier has to do. Sometimes customer requirements must be separated into several simpler items, if their logic is too complex to be understood or tested. Requirements received from the customer might have missing information or can be inconsistent with each other, the standard or legal frameworks. Without proper requirement analysis these problems would only be found during a later development step, which would make their correction harder and more costly.

The customer requirements were not the only source however. As mentioned before, the terminal control function went through a standardization process, the contents of these standards, be it an inside standard of the consumer or international standard, had to be taken into account at all levels of the development. It was important to identify the functions which interact with terminal control, for example the diagnosis service and the CAN controller, the customer requirements of these functionalities contained lots of useful information for the system documentation.



### 3.2.1 Analysis of the customer documentation

System requirements serve as the basis of the development for the other components [14]. To be able to fulfil this function, they must have the following characteristics defined by ASPICE:

- one object should contain only one requirement, keeping in mind that logically coherent parts shouldn't be separated
- requirement must be fully stated, it has to contain all the information necessary for implementation, and should always answer the question "Who should do what?"
- requirements should be atomic, their language should be as simple as possible, to help the understanding
- requirements should not contradict any other requirement
- requirement must be traceable, its sources must be accessible preferably through DOORS, the linking function of DOORS enables this
- requirements must be testable, they have to be defined in a way so that its contents can be verified

During the analysis of the customer documentation the system engineer can decide whether an item does fulfil these requirements or not. If they did not fully comply with the ASPICE rules above, I had to find a way during requirement formulation to correct it.

The first step of this process is to determine the purpose of the object. The purpose can be a use case or a higher-level requirement. By determining it we can focus on the problem rather than the solution, which may result in a "better" requirement, by letting the developer consider different alternatives for the implementation. On the other hand without the knowledge of the purpose, we cannot be sure whether the requirement makes sense, the requirement needs to be in the right context for the developer to be implemented. In my experience finding the purpose is usually the hardest part of the requirement analysis. The person who wrote the customer requirement concentrates most of the time on the problem's solution. This is of course a good approach if the customer can come up with a clear solution, but when he does not, then it becomes hard to guess what the problem was in the first place. During my work with the terminal control function I realized that the easiest way to understand the meaning of a customer requirement, which does not make sense in, is to look at its context. This includes where it is located in the document, what information items are close, and to simply imagine the

problem it tries to solve. Although this often helped, the best way was always consulting with the customer, and after clearing the problem providing him with a solution.

The next step is understanding the required action. First I had to select the key verb, which clearly states the action, e.g. “The BCM shall send KL15 signal” the action to be done is the sending. Sometimes the keyword can be hidden in the sentence as a noun, e.g. “The System shall allow storage of data” the storage word hides the real action which is to store. If the text contains more than one active verb it is possible that the customer object holds several different requirements.

The requirement has to contain the subject of the action. Understanding who the action’s subject and its object is, can be hard, if the text is in passive. In this case it might be helpful to transform the passive sentence into an active one, from which both the subject and the object is clear. Finding the action and the subject may seem marginal, but I often run into customer requirements, which grammatically had an action and a subject, but could not answer, that who should do what with them. A good example for this is the following requirement, “The BCM should not broadcast the KL-15 signal through LIN network”. Here the action is to “not broadcast”, the subject is the BCM, yet there is nothing to implement in the BCM, and no test can be provided.

Finally we can check for missing information with the following standard questions “What, Whom, When, How, How often”, if the developer cannot answer one of these questions, the requirement might need further clarification. Of course not all of them must be answered in each case, rather I had to decide which one is applicable for a given customer requirement.

### **3.2.2 Requirement formulation**

After understanding the content of the customer requirement I had to decide in which form to enter the requirement into the system documentation. The most frequent solution was as well-formed requirements sentences, but depending on the logics contained in the text sometimes it was more appropriate to use decision tables or state charts.

Decision tables are best used in case of complex logical expressions. Good examples are nested if-then clauses with several input parameters, or switch-cases with several output options. These tables are easier to interpret for the developer than long sentences with several parts. It is also easier to find any missing combination of in- or

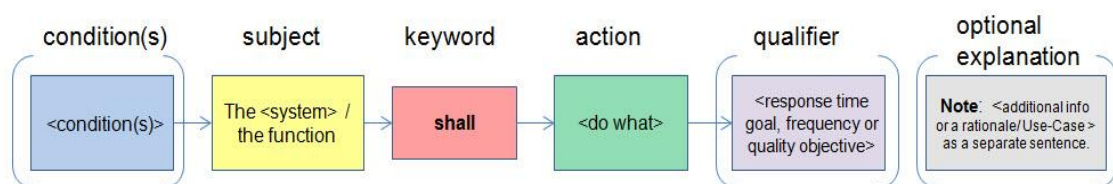
output values. A good example is Table 2, where it is much easier to showcase its logic in a table, because it would take several separate text requirements to write it down.

| Ubat                     | KL15 | Result of s_out  |
|--------------------------|------|------------------|
| $6 < U \leq 9 \text{ V}$ | ON   | "under voltage"  |
| $6 < U \leq 9 \text{ V}$ | OFF  | <initial value>  |
| $9 < U < 16 \text{ V}$   | ON   | "normal voltage" |
| $9 < U < 16 \text{ V}$   | OFF  | <initial value>  |
| $16 \text{ V} < U$       | ON   | "over voltage"   |
| $16 \text{ V} < U$       | OFF  | "over voltage"   |

2. table: Decision table of signal s\_out

State charts can be used to depict functional logics which are based on transitioning between states. These charts contain the possible states of the variables and the transition conditions between them. It is practical to use variable names on the charts, and insert the values of the variables as text after the diagram. This method is helpful when the values are changed, because only the text has to be modified accordingly, and the diagram can be left untouched. State charts are also an essential part of the system architecture design, and their creation will be described in the following chapter.

Well-formed requirements sentences are a way to represent requirements in natural language. They are technical statements about the product's behaviour and properties. In contrast to simple natural language they have to be clear, precise and as concise as possible. During the analysis of customer requirements the same or similar activities were made, then during the requirement formulation. In contrast to natural language well-formed requirement sentences have a strict structure. They have to match a given formulation template, which can be seen on Figure 11.



11. Figure: Requirement sentence template

The subject, keyword and action form the backbone of a requirement. They can be extended by optional parameters like conditions, qualifiers, explanation.

The subject always have to be specified even if it is clear from the context. For this purpose the usage of active sentences is mandatory. The subject can be for example “the function”, “the system”.

All object types have to be easily identifiable by the developers. This can be done by using a given keyword in each type, in case of requirements the keyword is “shall”, for information this can be “is” or “will”, for constraints “must”.

After the “shall” comes the verb describing the action. The main verb describes what exactly has to be done, because of this it has to be chosen carefully. The usage of adverbs or nouns as action verbs has to be avoided, for example “allow storage”. The action can be described in different level of detail, on higher system levels the verbs “allow” and “manage” can be sufficient, however in most cases verbs describing a certain action are preferred, e.g. “store, send, determine, debounce”.

Conditions can be present in requirements specifying the value of a signal or function logics. In case of complex logics the use of decision tables might be more appropriate. If the engineer decides to use requirement sentences, the emphasis is on the readability and clarity of the object. If the conditional statement is formulated well it resembles pseudocode, which helps the developer in implementation.

```
IF (AND-logic)
    the car ignition == 'ACTIVE'
    the car speed  >= '0'
THEN
    radio_voltage = 'ACTIVE'
ELSE
    radio_votlage = 'FALSE'
```

**12. Figure: A well-formed conditional statement**

The conditions should be written at the beginning of the object, if there are several conditions with the same logic, the logic operator can be specified in the beginning for the object. In my experience this is only useful when all conditions are linked with the same logic. If there are several outputs, it can be considered to divide them into separate objects, but only if they are not connected. Lastly while formulating the requirement the engineer should not forget to specify an ELSE branch.

Additional information can be added to the requirement as qualifiers. This information can describe for example the timing, frequency or tolerance of the values. During requirement formulation the usage of weak words should be avoided, such as “frequently, seldom”, this is important for creating test cases, since weak words cannot be tested. Making an educated guess, and later clarifying it with customer often times was the best solution.

Customer requirements frequently contain additional information, such as the description of the use case or purpose of the object, because these information might be needed to implement the function right. This information are obtained from the requirement analysis, and should not be omitted when they are not clear from the context. Although the requirement should always focus on what to do, these can be added to the object’s note field.

The BCM should determine the doors state from si\_innerdoor hardware signal.

Note: The door state shouldn't be determined from s\_innerdoor CAN signal, since that message has high latency.

**13. Figure: Note field embedded into the requirement**

### **3.2.3 Integrating the requirements into the documentation**

The system documentation does not solely consist of requirements texts. The documentation has to possess a structure which does comply with the rules of ASPICE, this structure contains headings and also the background information needed to understand the requirement texts.

First of all, the modules have to have a given structure, this structure can be outlined with the headings. Each module may contain several functions, these functions usually belong together because they interact with the same part of the car, for example the inner lighting or the terminal control. In terminal control there are two main functions: the first is responsible for calculating KL-15, the second is for KL-75. The modules should contain glossaries for acronyms and special words, these can help the developers in understanding the object texts. A good example are the terminal numbers, in the glossary I have included their use-cases. Each function starts with a description field, in this field the writer can give a short summary of what the function does, or can include the simplified version of the function logics.

After the description field comes the description of the function. Function components can differ greatly from each other, so I will mention the parts terminal control has. First the function initialization is defined, by giving the signals a starting value. After the initialization the requirements specify the actualization events or time intervals. The previous two chapters describe the function state, next comes the requirements dealing with the function logic. Some functions have software, hardware and mechanical components, these components can be separated into chapters. Since the terminal control function is safety critical it has components for error detection and error handling. Some of the signal values must be accessible from outside for car diagnostic, these signals are described in the error memory and the diagnostic parts of the module.

Each requirement must have given number of attributes, the value of these attributes can be chosen from a predefined list. Because the options are predefined, and usually the values are one word or a number, they can be sorted quickly, and can be used for creating statistics. These attributes are object type, component type, input signals, output signals, accessibility, ASIL classification, implementation and test status.

Although the keyword in the object text tells which object is a requirement, there is also an attribute for the same purpose. The object type attribute can be: information, requirement, heading and constraint. This attribute makes it easy to error check the database for traceability and test cases. The component type attribute as its name states assigns the object to a developer group, it can either be system, software, hardware or mechanical. In my experience this can be determined fast, but sometimes, especially in case of safety relevant features, it is not obvious whether it should be implemented in hardware or software.

On a system level we can't talk about variables, the objects carrying the information are called signals. They have strict naming conventions, and all of them must be entered into the main signal table. There are 3 kinds of signals: input, output and internal. Signals can later be defined in hardware and software. To implement a function, it is important to know the interface of the function. In the requirement texts variables must be named as signals, their text type is italic. Input attributes and output signals are entered into the corresponding attribute field. To maintain traceability according to ASPICE in the information flow I had to enter each signal into the central signal table, which stores all the signals used in the BCM. This attribute helps the developers to trace back a certain signal from the central signal table to the requirement.

The accessibility attribute defines the group which can access the vales of the function, these groups can be the driver, diagnostic services and developers. This kind of safety levels are getting increasingly important as the connected driving concept is rapidly evolving, it is getting more important to protect the BCM data from outside influences.

The final attribute is the ASIL classification. ASIL acronym stands for the Automotive Safety Integrity Level. It is a risk classification scheme. Its levels are described in the ISO 26262 standard, which defines functional safety standards for automotive electronics. There are 5 possible ASIL states: ASIL A, ASIL B, ASIL C, ASIL D and QM. From A to D the safety relevance of the function is increasing, QM means that the object is not safety relevant information. Usually the ASIL level is provided in the customer documentation, I just had to copy it from the customer documents.


While creating the system documentation of a function each requirement's traceability must be provided. For this reason DOORS objects can be connected with the help of links. I only had to connect the created objects with their source objects, mainly customer documentation. The developers connect their implementations with the system documentation objects. The linking process can become very time consuming when the function elements have lots of connections with other objects in different modules.

The previously mentioned attribute values come from either the requirement text or from the customer documentation attributes. However ASPICE rules require the traceability of the objects, so the system requirement should show whether it was implemented and tested. For this reason two attributes were created in the terminal control function. Originally each linked object should have been opened and their status checked to fill these attributes. Additionally when the connected DOORS objects would be changed the ones in the system documentation would have to be updated. I realized during the design of the terminal control document, that this repetitive process can be automated with a DXL script. I have written an extension script, which refreshes the attribute values automatically, when the terminal control module is opened. This script opens all the links connected to the requirement and checks if there is an implementation or a test implemented, then returns this value, and writes it inside the respective attribute fields. Of course the script would require the engineers to link each requirement with tests and implementations, and mark the status of these items, luckily the program described in Section 2 of my thesis checks exactly this.

To fully specify a function it is not enough to create objects only in its module. All signals had to be entered into the signal table. The terminal control function has additional requirements in connected modules, for example in modules describing the CAN interface, the error memory table or the module describing the sleep mode. These items were previously created by other system engineers, so I only had to prepare the entries in the central signal table.

### 3.2.4 Examples of requirement formulation

In this chapter I will give examples of a raw customer requirements, and describe the steps of their processing. Both the customer and the system requirements are only samples. The original customer requirements are marked with red, while the system requirements are marked with yellow.




The so\_KL\_15\_HW analogue signal is sent with a true value of 5 V.

**14. Figure: Customer requirement A**


Example A is a requirement, it specifies the voltage value of so\_KL\_15\_HW signal. It belongs to the hardware development, since the output is an analogue signal. The requirement should be processed, because it is not ASPICE compliant.

The main verb of the requirement is “send” or “provide”. Before the main verb the “shall” verb should be written to show that it is a requirement. From the original text the subject is missing, here the subject is probably the BCM. The customer specifies the voltage for the ‘true’ value. Instead of “true” a functional value should be used, such as “active”. Finally the customer did not specify the tolerance of the voltage, this should be clarified with him, in this case it turned out to be 3%.



The BCM shall send so\_KL\_15\_HW with an active value of 5 V  $\pm$ 3% .

**15. Figure: System requirement A**



The system shall store s\_doorstate\_error with triple redundancy.  
IF the system detects data even the slightest chance of corruption THEN it shall attempt to recover the data respectively from the other locations.

**16. Figure: Customer requirement B**



Requirement B is clearly a requirement, since it has the shall keyword, also has a clear subject “the system”, and the main verbs are easy to identify, these are “store”, “detect” and “attempt”. The problem is however, that the text contains several requirements.

The customer has specified 2 requirements in the text: s\_doorstate\_error should be saved, and if an error occurred it should be recovered. Since their contents are independent they should be separated into two objects. The second part of the customer requirement has redundant part, such as “slightest chance”, “attempt to recover” and “respectively”. The requirements should be concise, any redundant word should be left out. Finally the “other” qualifier should be changed to the exact location of the information.

---

The BCM shall store s\_doorstate\_error with triple redundancy.

---

The BCM should recover s\_doorstate\_error from the error memory in case of an error.

**17. Figure: System requirement B**

---

The storage of the signal door\_open\_HW is not allowed.

---

**18. Figure: Customer requirement C**

Here the customer text seems to be a requirement, it has an active word “not allowed” and states the behaviour of a signal. However to every single system requirement a test case must be associated, and since the customer text is a negative sentence, it cannot be tested, and it does not specify anything for the developer to implement.

Through requirement processing it becomes clear, that this is not a requirement, but an information object. It should be kept, for clarifying the context, but will not be implemented and tested. Also the word “allowed” hides the true main verb which is “to store”, it is easy to overlook the main verb when it is in adverb form.

---

NOTE: The BCM shall not store door\_open\_HW.

---

**19. Figure: System requirement C**

### 3.3 System architectural design

During the system architectural design I have created the system architecture of the terminal control function based on the system documentation, which was prepared in the previous step.

The system architecture is a graphical representation of the product's system model, although it does contain textual information. According to the ASPICE rules, both the system documentation and the architecture should be a complete and standalone representation of the system, and the bidirectional traceability between the two has to be provided.

The system architecture has several benefits compared to the system documentation. Since it is a graphical representation, it is easier to understand regardless of the language spoken by the developer. The visual representation also makes implementation easier, since the diagrams can easily be transformed to UML and from there into code. Architecture modelling programs provide several useful features, such as automatic code generation, automatic test generation and simulation of the system and performance analysis.

From the point of a system engineer the creation of system architecture is useful, because it allows the engineer to think through the whole logic and structure of the product, which makes it easier to find loopholes and errors in the customer specification. A complex product can have thousands of used variables, although in the system documentation for a given module there is always enough information concerning the variable, it is not enough to see a variable's significance on a system level, because a variable can be used in several modules for different types of logics. The system architecture puts great emphasis on visualizing the flow of the variables, it helps the engineer understand its significance. This is useful in case the customer requirements are changed, the effect of the change on the whole system can be seen through the variable flows present in the system architecture. This would be impossible, with the use of only a written documentation, in this case all modules connected with any logical operation would have to be reviewed for each change request.

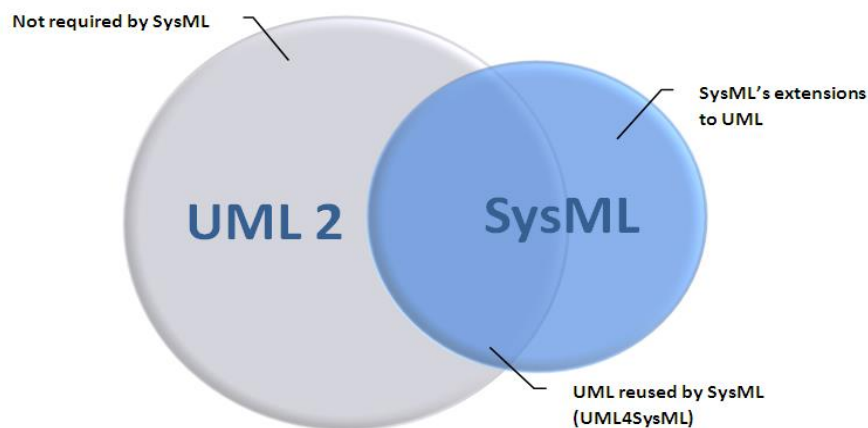
The system architecture can in later process steps be used as a source for the hardware and software architecture. The software architecture can take over entire components from the system architecture, if it is designed correctly. However I had to

keep in mind during its design phase that, most of the software functions can also be implemented in hardware, and by creating the system architecture with logics used for software, can later hinder the product development, because it restricts the developer.

The architecture has two main parts: static and dynamic. The static part describes the components, and the signals used and created, while the dynamic shows the behaviour of the system, and how the signal values are determined. The architecture was implemented in the SysML language. The Enterprise Architect tool is used for this purpose.

### 3.3.1 The SysML language

The System Modelling Language [15] [16], or short SysML, is a modelling language for system engineering purposes. The language is based on the Unified Modelling Language (UML), which is widely used for software architecture modelling.

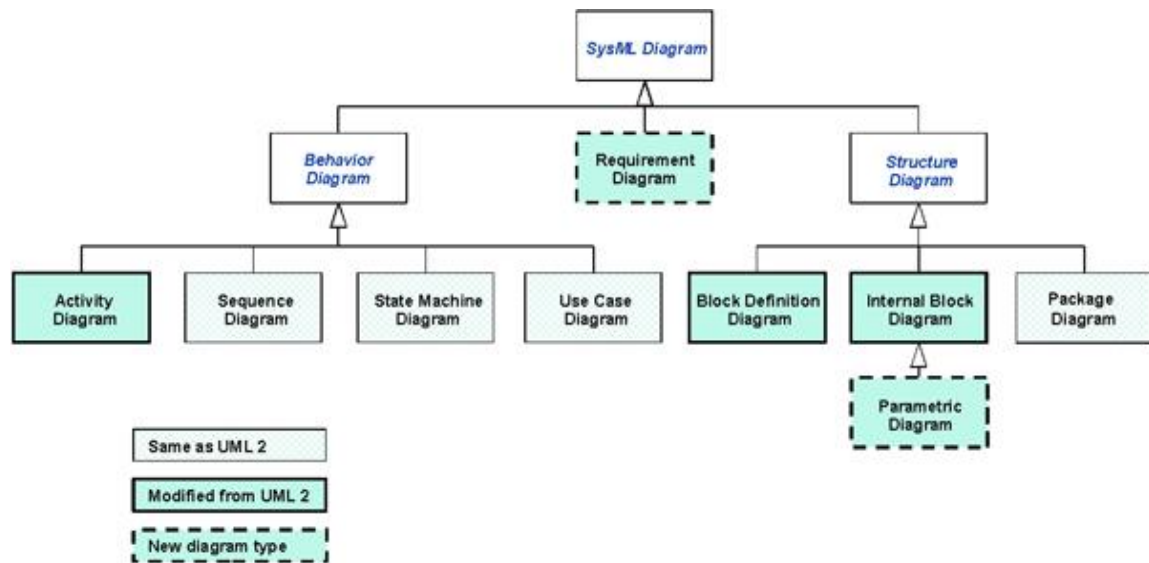


20. Figure: UML and SysML comparison [17]

SysML has some extension, which make it less software-centric compared to UML. It has two new diagrams, requirement and parametric diagram. Requirement diagram lets the user model requirements, and their connections with other requirements and components. Parametric diagrams make it possible to create performance and quantitative analysis. Thanks to these diagrams SysML is capable of modelling not just software, but a wide variety of components too.

The SysML language has 9 diagram types, compared to the 14 of UML, this makes it easier to learn for new comers. The diagrams have to categories: behaviour

diagrams for describing dynamic behaviour, and structure diagrams modelling the static architecture.



21. Figure: The diagrams of SysML [18]

SysML, just as UML, is based on object-oriented paradigms. Every folder, diagram and element on the diagrams is the member of the object oriented structure. Each element has a class, and can have several properties. Elements can have parent items, from which they can inherit properties.

All the diagrams are surrounded by a diagram frame. This frame can represent simply the drawing area, but in case of diagrams, such as block definition diagram, it is used as the interface of the system. In the top left corner the diagram header is located, it contains the diagram kind, model element type, the element name and the diagram name. The rest of the space inside the diagram frame contains the content, this is called canvas. The canvas is composed of two graphical elements: nodes and paths. Nodes are symbols which represent a model element, they can be blocks, ports and activities. Paths are edges connecting the elements of the canvas, they are responsible for visualizing the connections and information flows. Textual information can be added to the paths and nodes, depending on their type they can be notes, keywords and properties.

In the next subchapters I will introduce the main diagram types, and their ruleset, and provide an example from the terminal control function.

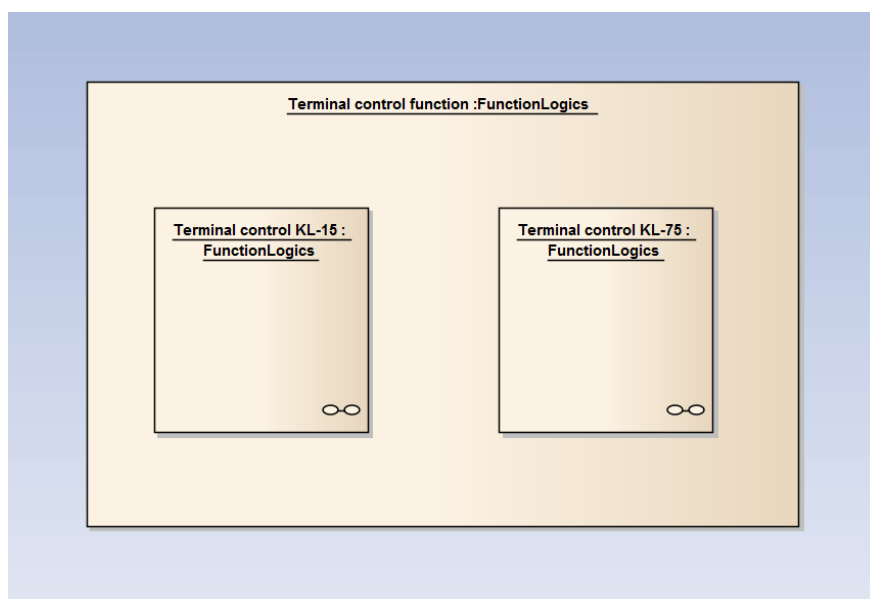
### 3.3.2 Package diagram

The system architecture is too complex to showcase its logics and components on a single diagram. To make the system architecture well-structured it was developed in a tree structure, where the level of abstraction corresponds to the depth of the tree.

This tree structure was created by the usage of composite elements. Composite elements contain full diagrams, these elements are marked with a symbol resembling a lemniscate ( $\infty$ ). The elements can contain diagrams different from the diagram type they are located in, this makes it possible to connect the structure of a composite diagram with the function logics of an activity diagram.

To help the developer's navigate through the tree structure, the package diagram was created. Package diagrams contain the packages of a function, these packages correspond to more detailed logics. The package diagram of the terminal control function contain two packages, these are the logics responsible for calculating KL-15 and KL-75.

In my opinion this diagram type is the easiest to design. The elements on it usually have no special properties, and their connections are sparse or completely missing. They can be derived directly from the system documentation. As a rule of thumb the name of the package diagram, and the parent component is equal to the module's name, for me it was terminal control. The blocks of the diagram are equal to the main functions of the DOORS module, they are usually on the top level of DOORS hierarchy. In case of the terminal function these are the calculation of KL-15 and KL-75 signals.



22. Figure: Package diagram of terminal control function

### 3.3.3 Block diagrams

Blocks are the fundamental modular units in SysML for describing functional and conceptual entities, they can be used for software, hardware or the operational environment of the product. Blocks are similar to UML classes. They are usually used in several diagrams of the system. A block is a type, which means, that it may have several instances with similar features. Blocks can be derived from each other, they can inherit the properties and features. They are depicted as rectangles divided into several compartments. These compartments hold the block features, the name is on the top of the block, and it is the only compulsory part.

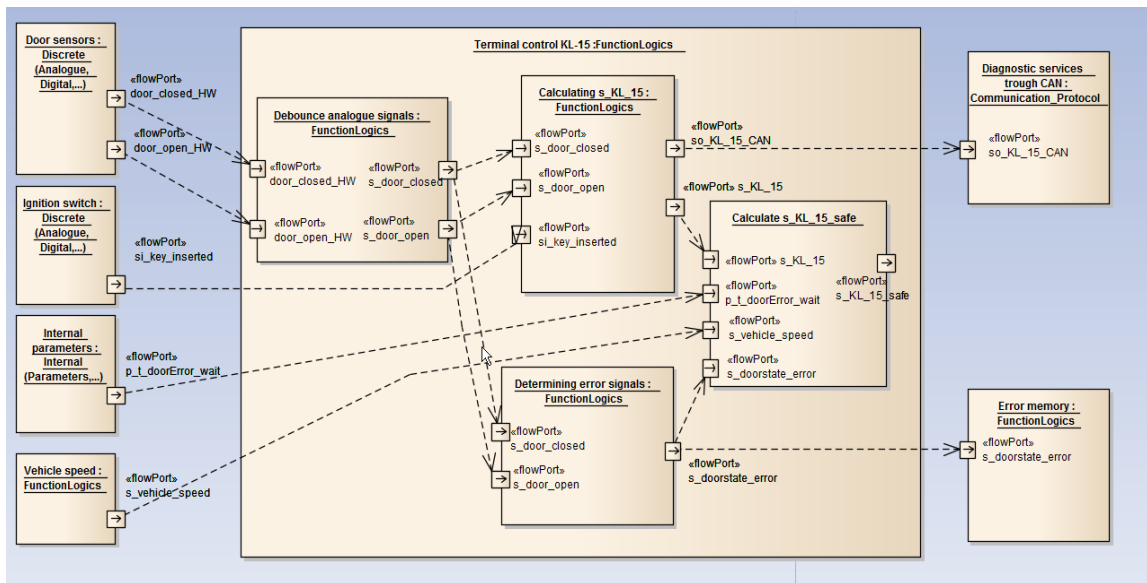
There are two kinds of block diagrams in SysML: internal block diagrams and block definition diagrams. The block definition diagram is a black box representation of a component. In case of this diagram the emphasis is on the components of the function, and the relation between them. It is called a black box representation, because the internal structure of the blocks is not provided in it. The internal block diagram on the other hand is a white box representation. It shows the internal structure of the component. Both diagrams mainly depict components and the interfaces and flows between them, the main difference is their scope.

If a component is placed inside another component, it means that it's the subsystem of the "bigger" block. One block can contain several blocks, which made it possible for me to depict the block composition. Composition can also be shown the same way as in package diagrams by the use of the lemniscate symbol, which means that the blocks structure is described on another diagram. In block definition diagram the lemniscate is used, while in internal block diagram usually the previous method.

I used the block diagrams not only for representing the system structure, but also for showing the developers the system interfaces. The item flows between systems can be information, control, streaming flows etc . Each signal has two ends, these ends are shown on the diagrams as squares, and they are called ports. Ports can have directions associated, either in-, out- or inoutput. This is an addition to UML, since in software usually the communication can be implemented bidirectionally, but SysML can be used to model thermal systems, where the flow direction cannot be reversed.

From the package diagram of the terminal control function, two block diagrams can be accessed, these diagrams belong respectively to the KL-15 and KL-75 calculation

functions. If the component does not have too many ports and components it is possible to use one diagram for both internal block diagram and block definition diagram. The block diagram I have created for the KL-15 function can be seen in Figure 23. It is also supplied as a supplement in a bigger size.



**23. Figure: Block diagram of function KL-15**

The diagram has 3 main parts: inputs, the inner structure of the function and outputs. The in- and outputs show the interfaces of the function. The function can have several radically different connections. In this case the function receives two hardware signals from the door sensors, these are door\_closed\_HW and door\_open\_HW. These signals are analogue, and must be processed both in hardware and software. It receives a digital signal from the ignition switch sensor, si\_key\_inserted, since this input is digital it can be processed from software. The function also communicates with other functions of the bodycomputer, the vehicle speed function supplies the s\_vehicle\_speed signal. With the help of the parameters the car manufacturer can tune the functionalities in the bodycomputer, the p\_t\_doorError\_wait is a time parameter, which influences the calculation of s\_KL\_15\_safe signal. However if it can be read by other control units or diagnostics, the interface must be shown on the diagram. This case one of the signals is saved into error memory, and the function also provides the value so\_KL\_15\_CAN through the CAN network. I chose the block diagram of KL-15 as an example, because it has all the major interface types.

The internal structure is shown in detail inside the big “Terminal control KL-15” block. There are 4 logical units in it: debounce analogue input, calculation of s\_KL\_15,

calculation of error signals, and calculation of `s_KL15_safe`. These logic units don't have to be formal subfunctions, they are just contain actions which are similar. They might hence not get a separate implementation, but can greatly help the work of the developer's understanding of the function. The internal blocks are currently not marked in the system documentation of terminal control, it was me who created these logical units. Three of the four units can be implemented from software, but the debouncing will inevitably have hardware components. This unit is also interesting since its inputs are analogue hardware signals, but the outputs are software signals inside the BCM.

The diagram shows the signal flows between the blocks. The flows are visualized by the ports and the dashed lines. The flow and port direction is marked with arrows. Although it is helpful to see all the signal flows it makes the diagram more complex, so usually the links are hidden, and only the ports are left visible. If the output signal of the function is only used inside the BCM the signal doesn't have to be connected to an interface, this is the case with `s_KL_15_safe`. BCM internal signal interfaces are usually not included, because several functions can use a signal inside, and it would make the creation of the diagram too long, if each signal's all interface would have to be found. However if a developer uses the internal signals, they have to connect these ports too with their diagrams.

After creating the block diagrams of the terminal control functions I had realized a few ways to automate their creation. Of course deciding the internal blocks, and the choosing which requirement belongs to which block cannot be automated. However after these have been defined and the requirements have been assigned with the help of a DOORS attribute we get a simple process. This is creating each block individually, then placing the ports for the signals on them, and finally linking the in- and output signals. After realizing this, I have written an extension in DOORS, this extension can be called inside the DOORS GUI, and it exports the internal blocks, their signals, and assigns them to an Enterprise Architect element. A separate script was written in EA for creating these elements, unfortunately the part which is responsible for linking the signal ports is not yet ready. The issue with this feature is that modifying the SysML properties of an element in EA is a problem which I could not yet solve as of yet. The created program will make the system engineering process faster, by solving almost completely one of the tasks of the developers.

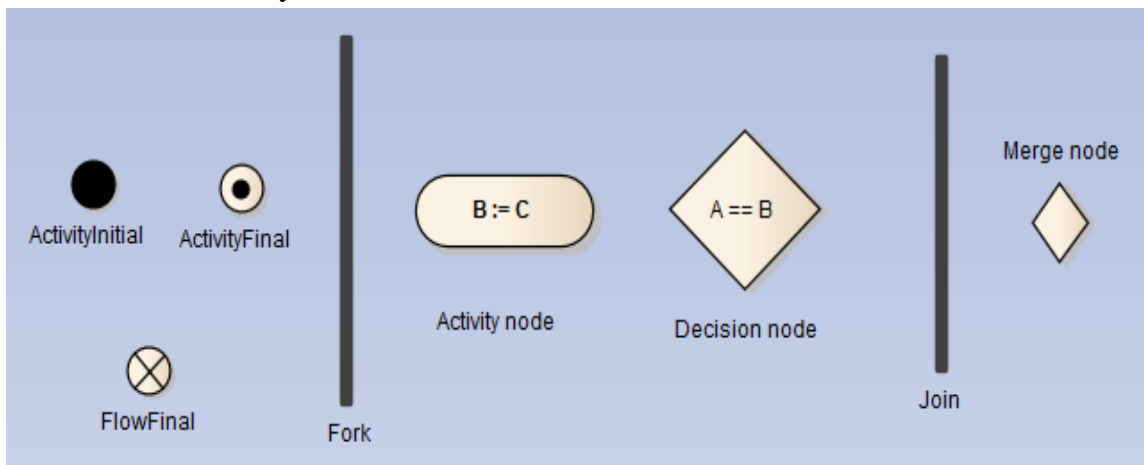


### 3.3.4 Activity diagram

Activity diagrams show the dynamic behaviour of the system. They can be regarded as flowcharts, where the symbol types and actions are restricted. They are graphical representations of workflow of actions, supporting choices and iterations.

The elements used in activity diagrams can be seen on Figure 24:

- initial and final node: they are the start and end of the workflow
- flow final node: it can be used to “destroy” tokens
- decision node: depending on the inputs provides an output on one of the flows
- merge node: any input flow is placed on the output flow, one input is enough for producing an output
- fork node: from one flow it creates several identical flows, used for input multiplication
- join node: from several inputs creates one output flow, only executes when there is input in all flows, used for parallelization of activities
- activity node: when it receives an activating signal it executes the activity

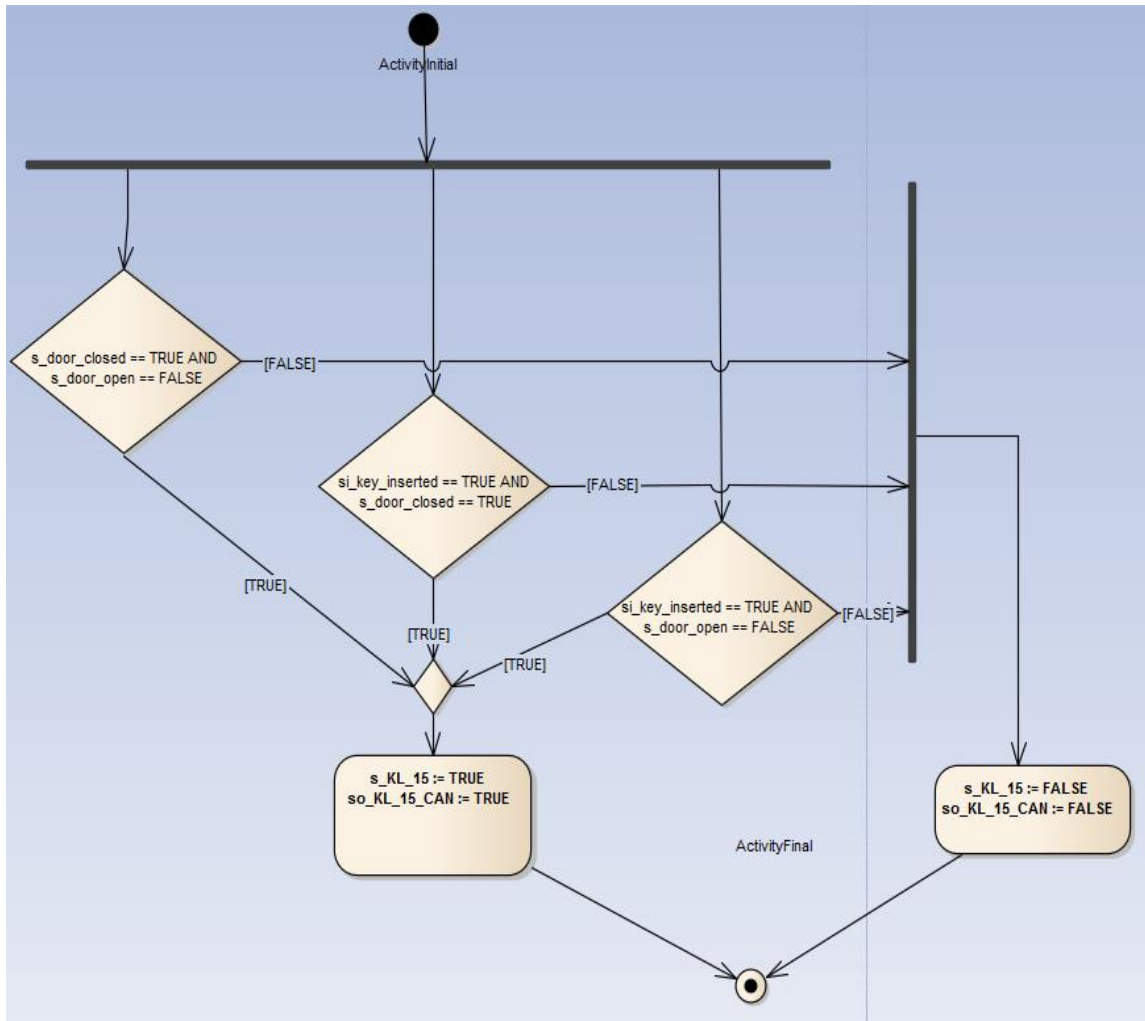


24. Figure: Activity diagram elements

Activities are the atomic elements of activity diagrams. They are responsible for creating the outputs, by setting the signal values or transforming the input flow. For example when the system is a grain mill, the input is the grain seed and the output is the flour of the “mill” activity.

The elements on the activity diagrams can be linked to show the item flows. In SysML the general unit of flow is the “token”, a token can be a signal value, analogue voltage or even thermal entropy. The flows always have directions, and they can be either signal or control flows. To be able to model physical systems SysML has a main rule for

flows, which is that no token can be duplicated or lost without an action. To fulfil this rule, the system engineer must use fork nodes for the multiplication of tokens, and merge or join nodes for assembling them.



**25. Figure: Activity diagram of s\_KL\_15 and so\_KL\_15\_CAN**

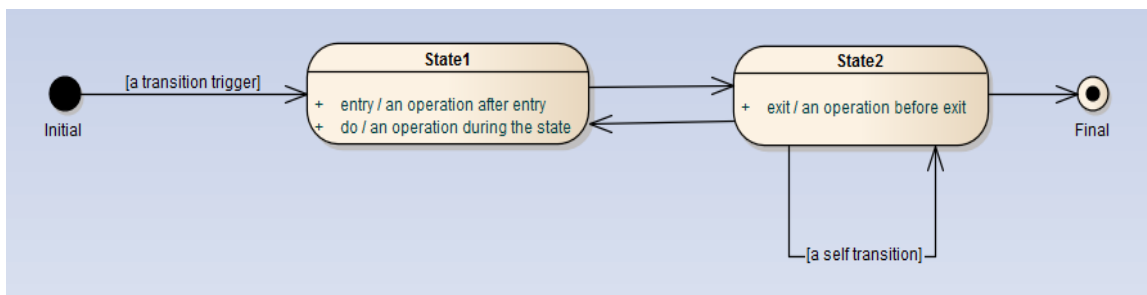
The activity diagram in Figure 25 shows the calculation of the signals s\_KL\_15 and so\_KL\_15\_CAN. It is a good example of activity diagrams, because it contains all the fundamental elements. The value of two signals is always identical, but they have different uses. The s\_KL\_15 signal is used inside the BCM, while the so\_KL\_15\_CAN is an output signal of the BCM, and it is transferred by the CAN interface to the other ECUs in the car. These signals are calculated from 3 inputs: s\_door\_closed, s\_door\_open and si\_key\_inserted. Their values are determined from a 2 out of 3 voting. The KL-15 signals are safety relevant, because they are linked to the car lighting and since they act as a power supply for other components, they can easily deplete the car battery. Their safe calculation is ensured by this voting system.

When creating an activity diagram it is important to look at the token flow. The voting is done by three different comparisons, where 2 signals are compared. From the initial node 1 token is supplied, but 3 comparisons has to be made, which requires fork node, this node multiplies the tokens. The comparisons are visualized by decision nodes, the 3 decision nodes are executed parallel, and depending on the decision an output token is supplied in either the true or the false line.

There are two possible outcomes, these outcomes are represented by two activities, where the function either set s\_KL\_15 and so\_KL\_15\_CAN to true, or false. If either of the three decision nodes return with a true value the signals have to be set to true, and the function should terminate. This behaviour is achieved by a merge node, which immediately transfers the received token to the output, so when one comparison is true, the activity is fulfilled, which in return sends a token to the activity final node. However to the signals to become false all comparisons must return with a false signal value. The join node must receive a token at all inputs to supply an output, so by connecting the three lines to it we can ensure that the activity is processed only when all the three comparisons are false.

### 3.3.5 State machine diagram

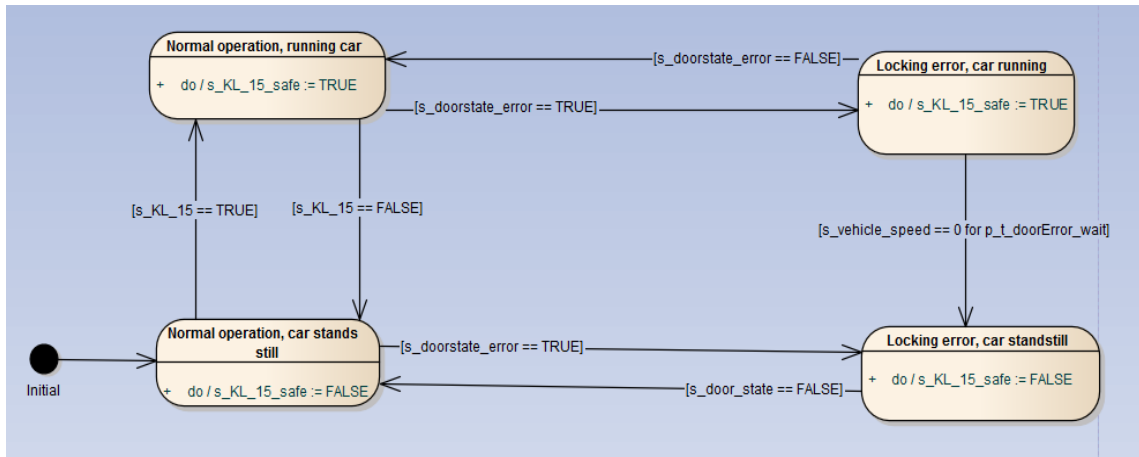
State machines are used for visualizing the dynamical behaviour of the system. They are used to represent the lifecycle of a block. The difference between activity diagram and state machine is, that the component's behaviour is influenced by its current state. For example the activity to stop a moving vehicle can be described by an activity diagram, but the actions which can be done with the car depend on whether the car is moving or it is standing still.



26. Figure: State machine diagram elements

Elements used in state machine diagrams are shown on Figure 26, these are:

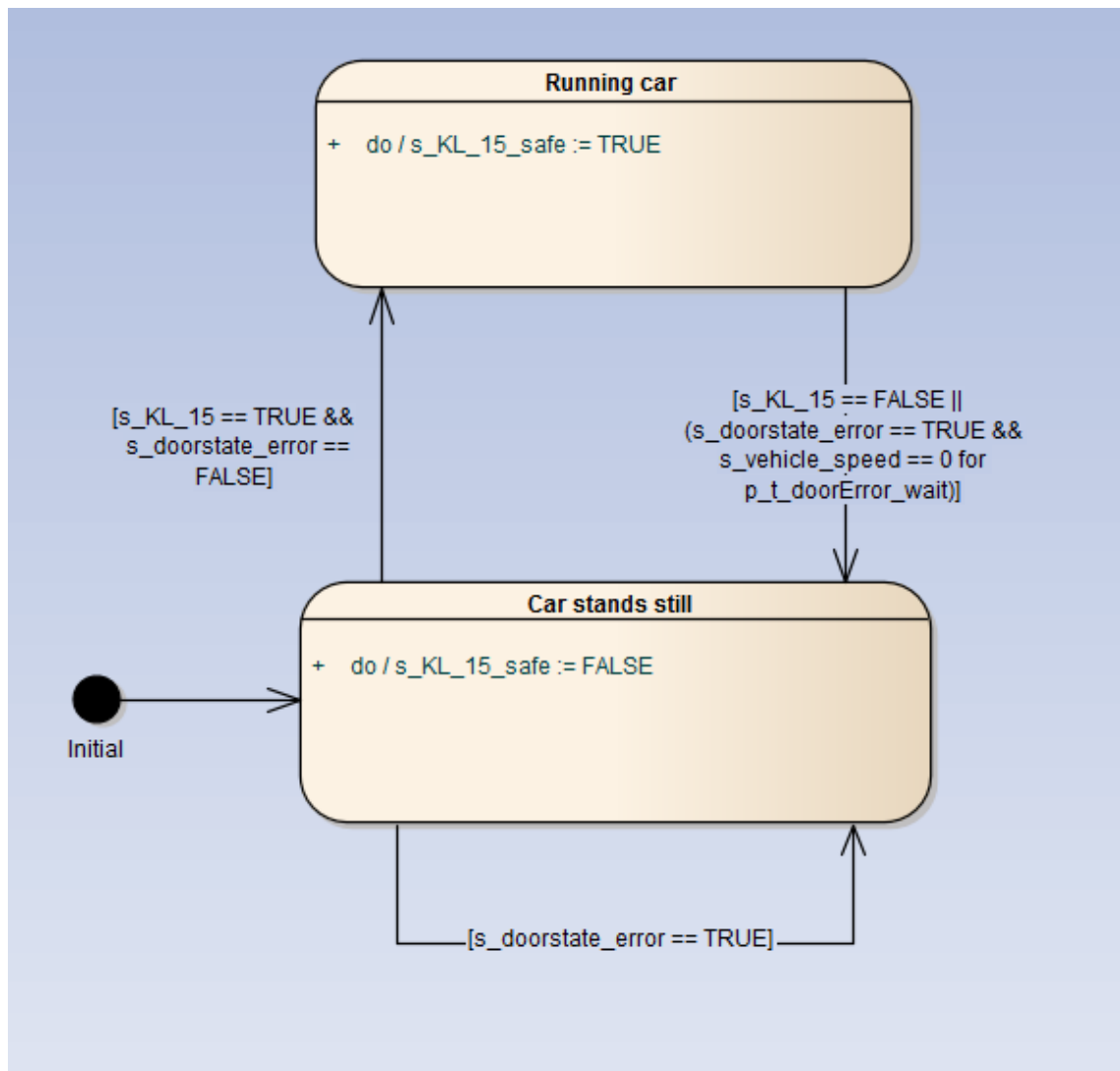
- initial and final node: the starting point of the system, final point of the system
- state node: represents an individual state
- transition: it usually links two states, however in rare cases a state is allowed to self-transition, the transition trigger can be a signal or call event, time event or change event
- operations: the operation the system does in a certain state, can be initiated after entering the state, inside the state or before exiting the state



**27. Figure: State machine of s\_KL\_15\_safe, complex**

The state machine on Figure 27 shows the calculation of the s\_KL\_15\_safe signal. This signal is a variation of s\_KL\_15, however it is used for higher ASIL classification functions. These functions should not turn on when there is an error with the key or door sensors, because then the safety of the passengers could not be provided. On the other hand if the error occurs during the travel it should only switch off after the car has stopped, and then the restart should not be possible.

The function takes in account two parameters: whether there is a locking error, and also the car speed. From these two parameters all 4 combination is possible, and these combinations serve as the 4 states, merged by the state nodes. The execution is started from the initial node, where the operation is normal and the car stands still. In normal operation mode s\_KL\_15 and s\_KL\_15\_safe is identical. When the s\_doorstate\_error switches to true, the operation mode changes to locking error. The operation can transition vice-versa between car standstill error and normal operation, but not to error and car standstill. Not all states can transition to each other. Inside the states we have a do operation which sets the value of s\_KL\_15\_safe.



**28. Figure: State machine of s\_KL\_15\_safe, simplified**

Figure 28 showcases the same logic as was described previously. However this representation of the system is simplified, uses only 2 states instead of the 4. It is easier for the implementing developers, however puts less emphasis on the inner working of the function. I included both versions into my thesis to show that usually there are several possible solutions for an activity or state machine diagram. This means that these diagrams cannot be generated with a program, and the experience and background knowledge of a system engineer is needed to design them.

## 4 Conclusion

During my thesis work I got introduced to the system engineering process and tools, and also developed extensions to help the work of the developers. I become acquainted with the rules of the ASPICE standard, and the development of automotive electronic components.

I have prepared the system documentation of the terminal control function. This can later be used as template function, which is helpful considering that almost all other components use the outputs of the terminal.

I have also written several programs and scripts extending the capabilities of IBM Rational DOORS. In my thesis Section 2 describes the design of an error checking tool in great detail, and its implementation in the DXL language. Two additional extensions are discussed in less detail, one automates the setting of attributes values, other the creation of block diagrams in the system architecture. During writing these extensions I have learned how to design more complex programs, and how to test them, additionally became familiar with the DXL language.

In the future I am planning to extend the capabilities of the error checking tool. I want to make it able to automatically correct the most usual errors, and finish the part responsible for sending notification messages in emails to developers of the errors connected to them. I will also finish the script responsible for creating the block diagrams in Enterprise Architect.

## 5 Table of figures

|   |    |
|---|----|
| 1. Figure: Error correction costs [4].....                                  | 8  |
| 2. Figure: Process dimensions according to the ASPICE Reference Mode [5]... | 9  |
| 3. Figure: Bosch Bodycomputer module [6] .....                              | 16 |
| 4. Figure: The DOORS user interface .....                                   | 17 |
| 5. Figure: The structure of DOORS legacy URLs.....                          | 19 |
| 6. Figure: The Enterprise Architect user interface.....                     | 20 |
| 7. Figure: The element definition window.....                               | 21 |
| 8. Figure: The levels of the development documentation.....                 | 23 |
| 9. Figure: The DXL interaction window .....                                 | 25 |
| 10. Figure: The Jenkins-CI user interface.....                              | 27 |
| 11. Figure: Requirement sentence template.....                              | 41 |
| 12. Figure: A well-formed conditional statement.....                        | 42 |
| 13. Figure: Note field embedded into the requirement.....                   | 43 |
| 14. Figure: Customer requirement A .....                                    | 46 |
| 15. Figure: System requirement A .....                                      | 46 |
| 16. Figure: Customer requirement B .....                                    | 46 |
| 17. Figure: System requirement B .....                                      | 47 |
| 18. Figure: Customer requiriement C .....                                   | 47 |
| 19. Figure: System requirement C .....                                      | 47 |
| 20. Figure: UML and SysML comparison [17] .....                             | 49 |
| 21. Figure: The diagrams of SysML [18] .....                                | 50 |
| 22. Figure: Package diagram of terminal control function .....              | 51 |
| 23. Figure: Block diagram of function KL-15 .....                           | 53 |
| 24. Figure: Activity diagram elements .....                                 | 55 |

|  |    |
|--|----|
| 25. Figure: Activity diagram of s_KL_15 and so_KL_15_CAN ..... | 56 |
| 26. Figure: State machine diagram elements .....               | 57 |
| 27. Figure: State machine of s_KL_15_safe, complex .....       | 58 |
| 28. Figure: State machine of s_KL_15_safe, simplified .....    | 59 |

## 6



## 6 Bibliography

- “Growing number of ecus forces new approach to cars electrical architecture,” 25 09 2012. [Online]. Available: <http://www.newelectronics.co.uk/electronics-technology/growing-number-of-ecus-forces-new-approach-to-car-electrical-architecture/45039/>.
- 1] “This Car Runs on Code,” 01 02 2009. [Online]. Available: <http://spectrum.ieee.org/transportation/systems/this-car-runs-on-code>.
- 2] “ASPICE model,” 01 08 2015. [Online]. Available: [http://vda-qmc.de/fileadmin/redakteur/Publikationen/Download/Automotive\\_SPICE\\_Process\\_Assessment\\_Model.pdf](http://vda-qmc.de/fileadmin/redakteur/Publikationen/Download/Automotive_SPICE_Process_Assessment_Model.pdf).
- 3] T. S. Duncan Seidler, “ASPICE Made Easy-Case Studies and Lessons Learned,” in *IBM Automotive Engineering Symposium*, 2013.
- 4] VDA QMC Working Group 13 / Automotive SIG, “Automotive SPICE Process Assessment / Reference Model,” 16 07 2015. [Online]. Available: [http://www.automotivespice.com/fileadmin/software-download/Automotive\\_SPICE\\_PAM\\_30.pdf](http://www.automotivespice.com/fileadmin/software-download/Automotive_SPICE_PAM_30.pdf). [Accessed 24 08 2015].
- 5] “Bosch Body Computer Overview,” 02 11 2015. [Online]. Available: [https://de.bosch-automotive.com/en/parts/parts\\_and\\_accessories/motor\\_and\\_systems/electronic\\_comfort\\_systems/body\\_computer\\_modul\\_1/body\\_computer\\_modul](https://de.bosch-automotive.com/en/parts/parts_and_accessories/motor_and_systems/electronic_comfort_systems/body_computer_modul_1/body_computer_modul).
- 6] “IBM Rational Doors,” 02 11 2015. [Online]. Available: <http://www-03.ibm.com/software/products/de/ratidoor>.
- 7] IBM Corporation, The DXL reference manual, release 9.6, 2014.
- 8] “Enterprise Architect,” 02 11 2015. [Online]. Available: <http://www.sparxsystems.com/products/ea/index.html>.
- 9]

- “Jenkins main page,” [Online]. Available: <https://jenkins-ci.org/>.  
10] [Accessed 05 11 2015].
- “Regular Expressions,” [Online]. Available:  
11] [https://en.wikipedia.org/wiki/Regular\\_expression](https://en.wikipedia.org/wiki/Regular_expression). [Accessed 05 11 2015].
- Microsoft Inc., “Ole object creation and manipulation,” [Online].  
12] Available: <https://support.office.com/en-au/article/Create-edit-and-control-OLE-objects-e73867b2-2988-4116-8d85-f5769ea435ba>. [Accessed 06 11 2015].
- “Terminal designations,” [Online]. Available: [http://www.bosch-automotive-tradition.com/media/en/automotive\\_tradition\\_1/teile\\_1/switches/downloads\\_3/klemmenbezeichnungen.pdf](http://www.bosch-automotive-tradition.com/media/en/automotive_tradition_1/teile_1/switches/downloads_3/klemmenbezeichnungen.pdf). [Accessed 09 11 2015].  
13]
- d. S. Chriss Rupp, Requirements Engineering & Management, 5th ed.,  
14] 2009.
- “Object Management Group,” 03 11 2015. [Online]. Available:  
15] <http://www.omg.org/>.
- A. M. R. S. Sanford Friedenthal, A practical guide to SysML, Croydon,  
16] UK: CPI Group Ltd., 2012.
- Y. Dajsuren, “Automotive System and Software Architecture,” 24 03  
17] 2015. [Online]. Available: [http://www.win.tue.nl/~aserebre/2IW80/2014-2015/20150324\\_AutomotiveArchitectures\\_Dajsuren.pdf](http://www.win.tue.nl/~aserebre/2IW80/2014-2015/20150324_AutomotiveArchitectures_Dajsuren.pdf). [Accessed 21 11 2015].
- “Modeling Structure with Blocks in SysML,” [Online]. Available:  
18] [https://inf.mit.bme.hu/sites/default/files/materials/taxonomy/term/445/13/09\\_CES\\_Structure-Modeling.pdf](https://inf.mit.bme.hu/sites/default/files/materials/taxonomy/term/445/13/09_CES_Structure-Modeling.pdf). [Accessed 23 11 2015].

7 Supplement

