MŰEGYETEM 1782

SZAKDOLGOZAT-FELADAT

Barta Gábor Kevin (G40RDH)
szigorló villamosmérnök hallgató részére

AUTOSAR I-PDU Multiplexer szoftvermodul tesztelése

A modern gépjárművek biztonságtechnikai és kényelmi funkcióinak megvalósításában, környezetvédelmi jellemzőinek javításában stb. egyre jelentősebb szerepet kapnak a számítástechnikai megoldások. Ma egy prémium személyautó gyártójának közel száz elektronikus vezérlőegységből (ECU) és számos fedélzeti kommunikációs sínből kell kialakítani egy megbízhatóan működő elosztott rendszert, amely komoly algoritmus- és kommunikációtervezési, illetve munkaszervezési kihívást jelent. Az így adódó komplexitás uralására alakultak ki különféle szabványok, pl. a megbízható kommunikáció biztosítására a CAN és FlexRay sínek, a valós idejű feladatok futtatására az OSEK operációs rendszer vagy a futási idejű monitorozást támogató XCP protokollcsalád. A vezető autógyártók által 2002-ben életre hívott AUTOSAR konzorcium célja az, hogy ezen szakterületi szabványokra építve specifikáljon egy (i) *alapvető szolgáltatásstruktúrát* amely eltakarja a hardver sajátosságait és támogatja az alkalmazási szoftver hordozhatóságát (base software stack, BSW), (ii) egy *modellezési nyelvet* az ECU-kon futó alkalmazási szoftver szabványos leírására (software component template), és (iii) az alkalmazások és BSW-k ECU-n belüli és ECU-k közti *transzparens kommunikációját* lehetővé tevő elosztott runtime szolgáltatást (RTE):

- A *base software stack* magában foglalja az alacsony szintű eszközmeghajtókat (pl. EEPROM és Flash driverek), az ezeket eltakaró absztrakciós rétegeket (pl. memória absztrakciós felület) és az ezekre ültetett magas szintű funkciókat (pl. perzisztens adattárolás).
- A *modellezési nyelv* lehetővé teszi, hogy precízen specifikáljuk az adattípusokat, illetve az alkalmazást alkotó komponensek interface-eit és belső felépítését.
- Az *RTE* egy generált glue kód réteg, amely eltakarja az alkalmazáskomponensek elől, hogy az általuk fogadott vagy küldött információ pontosan hogyan jut el a forrástól a célig, potenciálisan ECU-k közötti kommunikációs buszok igénybevételével.

A konzorcium jelentős hangsúlyt fektet az *API-k szabványosítására,* de kifejezetten támogatja a versengést az egyes szolgáltatások *megvalósításában* („Cooperate on standards, compete on implementation”).

A jelölt feladata az AUTOSAR base software stack egy moduljának tesztelése a kapcsolódó feladatok elvégzésével az alábbiak szerint:

- *A szabvány kapcsolódó részeinek megismerése:* (i) ismertesse az AUTOSAR rétegzett BSW struktúráján belül a *kommunikációért felelős modulok szerepét* és (ii) vázolja ezek együttműködését egy olyan *forgatókönyv bemutatásával*, amelyben két, különböző ECU-n futó alkalmazáskomponens vált üzenetet az I-PDU Multiplexer modul igénybevételével.
- *Tesztelés alapfogalmainak bemutatása:* (i) ismertesse a modulteszt szerepét a szoftverfejlesztésben, (ii) a fekete doboz tesztelés sajátosságait, (iii) a tesztesetek tervezését határérték analízis alapján, illetve (iv) mutassa be, hogyan ellenőrizhető fedettségi mérésekkel az, hogy a lehetséges lefutások tesztek által megvizsgált részhalmaza várhatóan jól reprezentálja a modult.

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék
1117 Budapest, Magyar Tudósok krt. 2. I. ép. I.E.444.
Telefon: 463-2057, Fax: 463-4112
http://www.mit.bme.hu • e-mail: mitadm@mit.bme.hu

- *Modulteszt implementálása:* végezze el az I-PDU Multiplexer modul tesztelését, ehhez (i) állítsa össze a tesztelő környezetet, (ii) hozzon létre különböző konfigurációs modelleket, (iii) tervezze meg és implementálja a teszteseteket és (iv) futtassa ezeket a különböző konfigurációk mellett. Ezután (v) elemezze a tesztek eredményét, majd tekintse át, hogy (vi) vannak-e olyan aspektusai az elvárt működésnek, amelynek való megfelelőséget a tesztkészlet vagy akár a módszer hiányosságai miatt nem lehet ellenőrizni.

Tanszéki konzulens: Dr. Sujbert László, docens
Külső konzulens: Szikszay László (ThyssenKrupp Presta Hungary Kft.)

Budapest, 2015. október 7.

……
Dr. Jobbágy Ákos
tanszékvezető

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Barta Gábor Kevin

TESTING OF AUTOSAR I-PDU MULTIPLEXER SOFTWARE MODULE

BELSŐ KONZULENS
Dr. Sujbert László
BME MIT

KÜLSŐ KONZULENS
Szikszay László
ThyssenKrupp Presta AG

BUDAPEST, 2015

Table of Contents

HALLGATÓI NYILATKOZAT

Alulírott **Barta Gábor Kevin**, 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. 12.

…….

Barta Gábor Kevin

Összefoglaló

A szigorú autóipari elvárások miatt az ECU szoftverek intenzív fejlesztési és tesztelési folyamatokon mennek át. Az én feladatom az I-PDU Multiplexer modul tesztelése volt, ami az AUTOSAR kommunikációs verem egyik Basic Software modulja.

A feladatomhoz hozzátartozott az AUTOSAR szabvány megismerése, főként a kommunikációért felelős modulokra koncentrálva. A munkám során megismerkedtem a szoftvertesztelés alapfogalmaival, mint például a fekete doboz tesztelés és kódlefedettség vizsgálat. Az I-PDU Multiplexer interfészeinek leírását és a modulhoz tartozó követelményeket is meg kellett ismernem. Elsajátítottam egy modellező eszköz használatát, melynek segítségével többféle konfigurációt lehet létrehozni a tesztelés alatt álló modulhoz. Ezzel a tudással kezdtem el a teszt környezet felépítését és a tesztesetek implementálását.

A tesztelt funkcionalitás alapján összetartozó teszteseteket külön tesztkészletekben helyeztem el. Így készült el három tesztkészlet számos tesztesettel, melyek megvizsgálják a modulhoz előírt tesztelhető követelményeket. A tesztesetek pontos dokumentálása és az egyes követelmények tesztjének forráskódhoz rendelése is szerves része volt a feladatomnak.

Az összes tesztelhető követelményt sikerült ellenőrizni a megírt széleskörű tesztesetekkel. Több konfigurációt alkalmaztam annak érdekében, hogy megbizonyosodjak a tesztelt modul viselkedésének helyességében. A tesztelés során a modul elérte a száz százalékos elágazás lefedettségét. Az előbb felsoroltak alapján a modul tesztelését befejezettnek lehet tekinteni.

Abstract

Due to the rigorous requirements dictated by the automotive industry, ECU software must undergo intensive development and testing procedures. The AUTOSAR standard dictates the requirements that ECU software modules must fulfill. The assignment was the software testing of the I-PDU Multiplexer module, which is a Basic Software module located in the AUTOSAR communication stack.

To successfully test the I-PDU Multiplexer module, an understanding of the AUTOSAR standard was necessary, with special emphasis on modules responsible for communication. Utilizing software testing principles, such as black box testing and code coverage analysis, and becoming familiar with the interfaces of the I-PDU Multiplexer was essential for testing of the module. To be able run extensive tests, a ThyssenKrupp Presta AG developed modeling tool was used to create several configurations in order to assemble diverse and distinct parameters for the module. Using the above mention knowledge, the construction of the testing environment and the implementation of the test cases could begin.

The test cases were categorized into separate test suites according to the tested functionality. Three distinct test suites with several test cases, were used to effectively examine the testable requirements of the module. The documentation of the test cases and the tagging of the requirements were also an essential part of the assignment.

All the testable requirements were successfully inspected, owing to the comprehensive test cases. Consequently, 100% branch coverage was reached and all the testable requirements were met. The module is deemed fully tested by requirements set by AUTOSAR and the Quality Management constraints set by ISO 26262.

1 Introduction

1.1 Background

Over the first decade of the new millennium, the rise of embedded systems in the automobile industry grew rapidly. With increasingly more complex electrical systems, such as autonomous cruise control (ACC), and safety-critical components, such as power steering systems, being installed into vehicles, the ECUs that control such systems, must undergo an extensive development cycle.

The development cycle of an ECU follows the V-model with many phases of verification and validation. The V-model will be discussed in more detail later. To guarantee the upmost quality, Tier 1 suppliers started to devote more time for the testing of both the hardware and software of ECUs, especially ones dealing with safety-critical systems. Following standards such as ISO 26262 and AUTOSAR, have also became a priority for manufacturers of the automotive industry.

1.2 Purpose

The main purpose of the AUTOSAR standard is to create a common foundation of ECU development, yet still promote competitions between manufacturers. The layer model of AUTOSAR promotes the idea of hardware independent implementation, thus attaining software transferability. Entire basic software stacks can be relocated onto different hardware platforms due to the AUTOSAR standard. To achieve this software transferability, AUTOSAR releases detailed specifications and precise requirements that ECU software developers carefully follow to implement in their software modules.

After implementation, the next stage is module testing. Modules are thoroughly tested to examine if the requirements are reached. Testing is done by using a framework, such as CUnit, that provides a well structured environment in order to test the modules. After proper documentation, a requirement analyzing tool runs through the software to check the location of each tested requirement. Aspects such code coverage are also benchmarked to measure if every functionality of the module was executed during testing. The entire testing process is to ensure software quality and to guarantee that the module lives up to the AUTOSAR standard.

1.3 Objective

My objective in the development process of ECUs, here at ThyssenKrupp Presta AG, was to test the Interface Layer Protocol Data Unit Multiplexer (IpduM) module, a BSW module of the AUTOSAR communication stack. The requirements were specified by AUTOSAR 4.0, and amended by ThyssenKrupp Presta AG. My job was to write test cases, with the help of the CUnit testing framework, to examine the functionality of the module and check if the requirements are met.

First step I had to take, was to understand the IpduM module by reading the specification released by AUTOSAR 4.0. The 4.0 is the release number of the AUTOSAR standard. After acquiring thorough knowledge of the module, I started to write test cases assessing the boundary values of the functions. Afterwards, I wrote test examining simple development error requirements, such as invalid parameters and uninitialized module.

In the next step, I tested the reception side of the IpduM module to examine if the I-PDUs are received and processed properly.

Afterwards I examined the transmission side of the module, to test the module for proper transmission of I-PDUs. The testing of transmission side was more difficult, because of miscellaneous requirements such as Just-In-Time updates and timed transmission confirmations.

Tagging the requirements and documenting the code according to Doxygen was also part of my work. This stage is probably the most important because proper documentation is essential for the requirement analyzing tool to process the tested requirements and to demonstrate the thought process of the test cases.

After finishing my work, the testing environment I prepared will go into code review for examination to check if it meets company standards. Afterwards it will be sent back to me if any corrections must be made.

2 Automotive Industry Standards

2.1 AUTOSAR

2.1.1 Introduction

AUTOSAR is a standardized software architecture developed by automotive original equipment manufacturers (OEM) and suppliers. The name AUTOSAR is derived from the first few letters of the words AUTmotive Open Systems ARchitecture. The goal of AUTOSAR is to create a reference architecture for automotive software development, due to the increasing complexity of software in modern automobiles.

2.1.2 Initiative

AUTOSAR was developed in 2003 by leading automobile companies, original equipment manufacturers (OEM), supplier, and various companies in electronics industry. The main driving force behind developing the AUTOSAR standard, was to create a foundation of basic functions, while offering a platform to encourage competition in the industry. This mentality is conveyed in the slogan, "Cooperate on Standards, Compete on Implementation." The goals of AUTOSAR are the following: [12]

- "Definition a reference architecture for ECU software"
- "Standardization interfaces between functions of the application software"
- "Transferability of software"
- "Collaboration between various partners"
- "Differentiate between hardware and non-hardware specific components"

2.1.3 Layer Model

The aim of AUTOSAR is to have the functions of the ECU be hardware independent and divided into atomic software components (SW-C). On an abstract level, the communication between software components is represented by the

Virtual Function Bus (VFB). The purpose of representing the communication with the VFB, is to indicate that the software components are independent of the ECU hardware. The reason for the layer model is that it simplifies the porting of software to different hardware. Due to AUTOSAR, all that needs to be done for example, when using a different microcontroller, is substitute the old microcontroller specific modules with the new ones. AUTOSAR divides the software architecture of an ECU into three abstract layers. [1] [12]

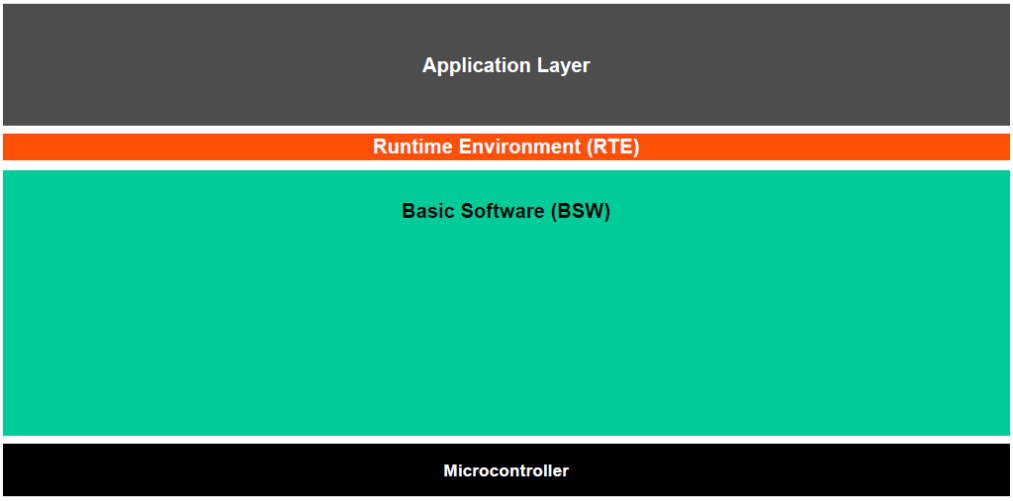

Figure 2.1 AUTOSAR Layer Model (Top View) [1]

2.1.3.1 Application Layer

At the top of the AUTOSAR layer model is the application layer. Located in this layer are the entirely hardware independent software components of the ECU. AUTOSAR does not specify the function of the software components, it only specifies the communication between the components through their interfaces. The application layer only communicates with the Run Time Environment (RTE), the layer underneath it. [1]

2.1.3.2 Run Time Environment (RTE)

The Run Time Environment embodies the communication between the Application Layer and the Basic Software Layer. The RTE is generated by the description file of the ECU, therefore it is dependent on the hardware. But by creating the RTE hardware dependent, its top interfaces linking to the Application Layer can be made hardware independent. [1]

2.1.3.3 Basic Software

The Basic Software is made up of atomic software modules that fulfill specific functions. The Basic Software can also subcategorized into different layers, depending on their purposes in the architecture. Starting from the upper layers of the Basic Software and heading down to the lower layers, the layers are more integrated into the hardware of the ECU. [1]

- **Services Layer:** The operating system and other services such as memory management, network services, and bus communication services are handled by this layer.
- **ECU Abstraction Layer (ECUAL):** Provides an interface for the functions of the ECU. This is independent of whether or not the functions are operated by drivers of the microcontroller or external devices. This layer does not depend on the microcontroller in use, however it depends on the ECU's inner structural design.
- **Microcontroller Abstraction Layer (MCAL):** The peripherals, such as memory, communication, I/O of the microcontroller are accessible by the drivers provided by this layer. The MCAL is highly dependent on the microcontroller in use.
- **Complex Drivers:** Complex functions not found in other layers are implemented here. This layers has direct access to the microcontroller, and is dependent on both the ECU and the microcontroller.

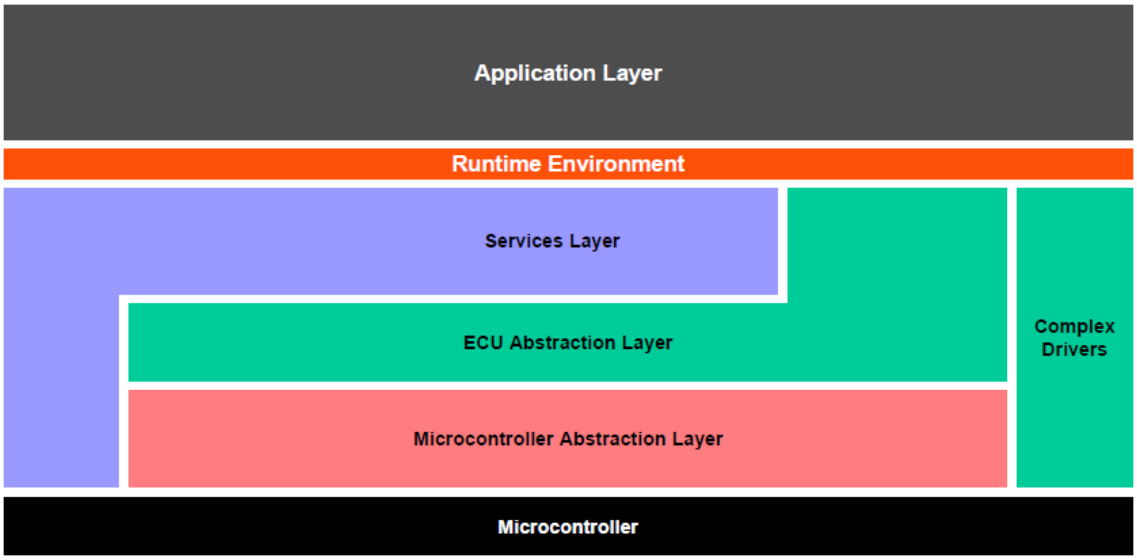

Figure 2.2 AUTOSAR Layer Model (Coarse View) [1]

2.1.4 Interface Types

In general, an interface is the boundary between two entities, where they can communicate with one another and exchange data. AUTOSAR categorizes three different types of interfaces. [1]

2.1.4.1 AUTOSAR Interface

An AUTOSAR Interface is an interface that enables communication through the Run Time Environment, between SWCs or between Basic Software Modules. An AUTOSAR Interface is specific to the application it is used by, and is generated with the Run Time Environment. Because of this, one side of an AUTOSAR Interface always connected to the RTE, and the other side communicates with the SWC or BSW module.

2.1.4.2 Standardized AUTOSAR Interface

A Standardized AUTOSAR Interface is a specific AUTOSAR Interface that the AUTOSAR standard defines. Through the Standardized AUTOSAR Interface, the SWCs reach AUTOSAR services, such as the Diagnostic Event Manager.

2.1.4.3 Standardized Interface

A Standardized Interface is an Application Programming Interface (API) defined in a programming language, usually C. This interface is also standardized within AUTOSAR, however it behaves differently as the AUTOSAR Interface, as it is not

generated with the RTE. Basic Software modules communicate with one another using the Standardized Interface. Communication between BSW modules and the RTE is can be provided through the Standardized Interface, but in this case the communication must be limited within the ECU.

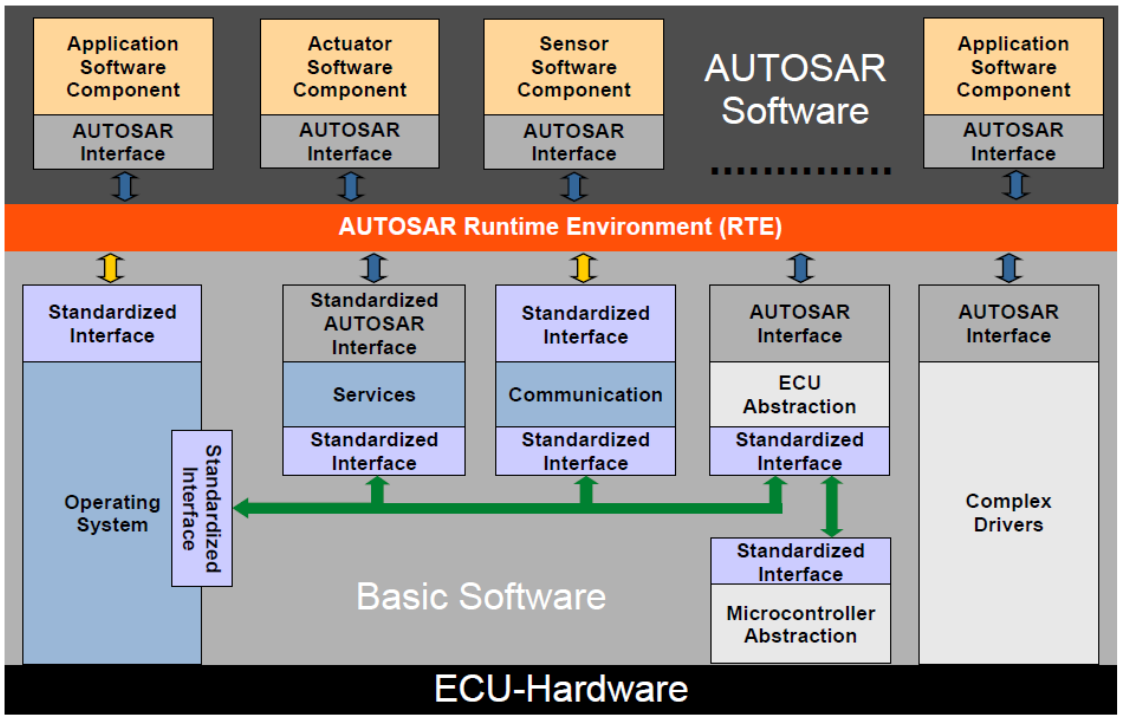

Figure 2.3 AUTOSAR Components and Interfaces View (Simplified) [1]

2.2 ISO 26262: Road vehicles – Functional safety

2.2.1 Overview

The automotive industry is increasingly becoming more dependent on electrical and digital systems. Due to the dependency of these systems, a standard had to be created regarding safety critical components. With this in mind, ISO 26262 was published in 2011. By applying these standards during the design and production of electrical systems, the safety aspect of products have improved.

The parent standard of the ISO 26262 is IEC 61508, which is a basic functional safety standard that can be applied to every industry. Whereas ISO 26262 is a standard specifically pertaining to automotive electrical and electronic (E/E) systems. [8] [11]

2.2.2 Development Cycle of Products

ISO 26262 specifies recommendations on developing safety-compliant products, through every stage of its development. The standard can be applied starting from the early conceptual design, and through phases such as, development, production, operation, service and even the final decommission of the product. At every stage of the product, numerous safety measures and audits are done to assure the proper utilization of the standard. [8]

2.2.3 Automotive Safety Integrity Level (ASIL)

ISO 26262 defines a system of classifying the product by the necessity and strictness of applying the safety standard. ASIL is calculated using three factors; severity, exposure, and controllability. [11]

- **Severity:** The severity of the failure, is the harshness of the outcome if it occurs. In other words, how much loss the failure can cause, to the user or the product itself.
- **Exposure:** The exposure is the probability of the failure happening.
- **Controllability:** The controllability of the outcome, is how much influence the user has on preventing damage or loss after the failure has occurred.

$$ASIL = Severity + Exposure + Controllability$$

There are five levels of ASIL ranging from ASIL-D, where the highest degree of assurance is needed because these are applications with high level of risk if failure occurs, to Quality Management (QM), which are applications where there are no automotive hazards involved. The five levels of ASIL are, in increasing order of safety requirements; QM, ASIL-A, ASIL-B, ASIL-C, and ASIL-D.

2.3 MISRA C

2.3.1 Purpose

The C language has become the dominant programming language used in embedded systems, due to run-time efficiency, low-resource requirements, and hardware flexibility. However the C language has its drawback, including loose syntax

requirements and restricted run-time checking. Due to the likelihood of inexperienced programmers making mistakes, which stem from the disadvantages of the C language, a standard was established when programming embedded systems. [5]

2.3.2 History

Motor Industry Software Reliability Association was established by automobile manufacturers to set guidelines for developing automotive embedded systems in the C language. These guidelines, named MISRA C were first published in 1998. Since then, two other editions were released in 2004 and 2012, named MISRA-C:2004 and MISRA-C:2012 respectively. [5]

2.3.3 Guidelines

The newest edition of the MISRA C guidelines include a total of 143 rules and 16 directives. The rules are accurately defined requirements, and are grouped into three categories; advisory, required, and mandatory. Two concepts differentiate between the three types of rules, compliance and deviation.

- Compliance is the degree of how intensely the rule must be followed or obeyed. In another words the obligation to the rule.
- Deviation is to what degree the rule can be strayed from.

The requirement of the above mentioned traits, decide which out of the three categories does a rule fall in.

- An advisory rule is one that both compliance and deviation is optional. Meaning it is not required to follow advisory rules, but it is recommended by the MISRA C guidelines.
- A required rule is one that compliance is required, unless it is justified by a deviation. Meaning if due to a reasonable justification stemming from the application of the software, it is acceptable not to obey the rule.
- A mandatory rule is one that compliance is required, and under no circumstances can the rule be omitted.

Directives are not as easily defined as rules, but are procedures that should be obeyed the same way as rules. The main difference between them is rules are well defined, while directives are subject to interpretation. Due to sometimes unclear

distinction between rules and directives, an example is given. “Precautions shall be taken in order to prevent the contents of a header file being included more than once,” is a directive while, “All object and function identifiers shall be declared before use,” is a rule. [3]

3 Software Testing

3.1 Introduction

In general, software testing is the act of evaluating if a software component or an entire system, meets the predefined requirements. The software undergoing testing has to meet certain requirements, in order ensure the proper behavior of the software. Even after extensive testing it cannot be guaranteed that the software works properly hundred percent of the time. So the purpose of testing is to increase the software quality of the modules. [10]

Another reason why software testing is important, is to locate errors and bugs in the software. These abnormalities usually originate from design errors during the development of the software, and they only surface during testing.

3.1.1 The V-Model

The V-model is a model representing the development process of software. The left branch, called the verification phases, of the V-model shows the steps the developer must take to create the software. While the right branch, called the validation phases, shows the steps of the tester must take to confirm the proper behavior of the software. [10]

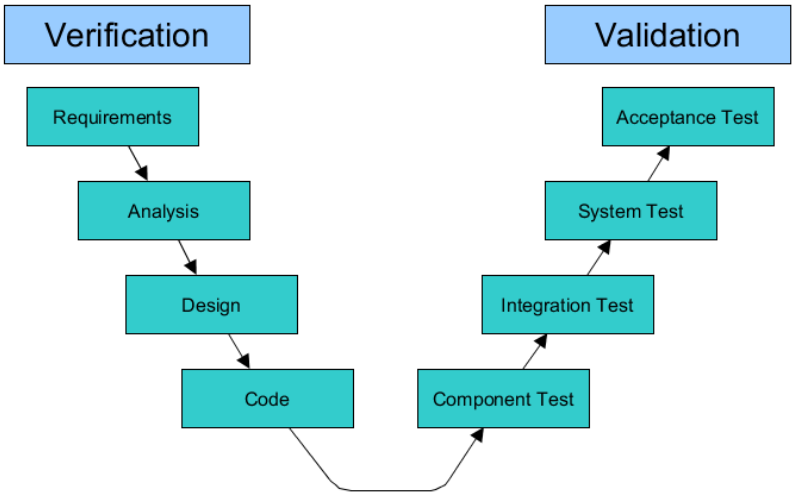

Figure 3.1 V Model of Software Development

Each verification stage must corresponds to a validation stage. Verification is the process of reviewing and analyzing the code without executing it. Validation is testing the software by executing the code.

The four stages of validation are explained in more detail.

3.1.1.1 Component Test

At this stage, the smallest individually testable components or modules are tested. The functionality of the component is being tested.

3.1.1.2 Integration Test

After the individual component testing, the components are combined and tested as a group. The goal of integration test, is to examine the behavior of the interface and the communication between the components.

3.1.1.3 System Test

After the integration test, the entire software undergoes testing. At this stage, the goal is to examine if the system fulfills the requirements.

3.1.1.4 Acceptance Test

The acceptance test is conducted by the end-user of the product, to check if the software qualifies and fulfills their expectations.

3.1.2 Complexity of Software Testing

Due to the complexity of the software it is difficult to run extensive and comprehensive test. For example, even taking a simple program, such as one that shall add two 16 bit integers together, has 2^{32} test cases, which can take ample amount of time. For actual software components in the industry, which are far more complex than the example given above, testing for all the possible outcomes is near impossible. Because of this, usually only boundary value analysis test are done to test outcomes, which will be explained in more detail when discussing Black-box testing.

3.2 Methods of Testing

There are several different techniques and methods of software testing. These testing methods can be classified into different categories depending on the purpose of

the test. Such classifications are correctness testing, performance testing, reliability testing, and security testing. Only correctness testing is explained in detail below. [10]

3.2.1 Correctness Testing

Testing for correctness of a software is the minimum requirement. This is usually the main purpose of testing. Essentially it is testing the software for a correct input given a correct output is produced. To test for correctness, a knowledge of some type of specification is needed, in order to tell correct responses from incorrect responses. This does not necessarily require the knowledge of the inner workings of the software. When the test is conducted with the understanding of the inner structure of a software it is called white-box testing, if the test is conducted with only the knowledge of the interface of the software, it is called black-box testing.

3.2.2 Black-box Testing

When testing a software with the black-box testing method, only the input, outputs, and specifications are known. It is called black-box testing, because the component is seen as a black box. The inner structure of the software is not seen from the outside. The tester only has knowledge of the interfaces of the software component, and not its inner structure. The main reason for using this technique, is to test the functionality of the component. Functional testing refers to performing the functions of the software and examining the input and output data. During testing, numerous inputs are processed, and the outputs are compared with the proposed results indicated in the specification. Only the specification is used to check for the correctness of the component.

The further we test different input values of the input space, the more we can assure ourselves of the quality of the software. However, conducting test with the entire input space is not feasible. Therefore a method called partitioning is used. Partitioning is assuming that the input values of a particular partition are equal. Meaning only one case from each partition is sufficient to test the entire input space. Correctly choosing one, usually more cases, from a partition, to represent the entire domain, is the basis of boundary value analysis. Boundary value analysis is a technique using boundary values as inputs for test cases, to test the entire partition. The boundary values are values on the on the border of the partition, usually the minimum and maximum values.

3.2.3 White-box Testing

During white-box testing, the component under testing is viewed as glass-box, meaning the entire inner architecture is seen by the tester. Using this technique, the test cases are constructed, with the structure of the component in mind. Aspects such as programming language, logical structure, and data flow are considered for the test cases.

Since the code of the module is known, it is entirely feasible to be able to fully traverse the code space, including executing every line of code (statement coverage), entering every branch (branch coverage), and covering all possible logical combinations (multiple condition coverage). Because of the extensive code coverage, "dead" code, code that is never executed, can easily be located.

There are times when testing methods, cannot clearly be classified as either black-box or white-box testing. The reason being, it is not always clear to what extent of looking under the hood of the software is considered white-box testing, rather than black-box testing. For example, the knowledge of the programming language used or the structure of the code might already infringe on the definition of the black-box testing method. Since the boundary between black-box and white-box testing is sometimes blurred, a combination of using testing tactics from both methods is called gray-box testing.

3.3 Code Coverage

Code coverage is the percentage of code that is effectively being executed during testing of the program. It shows the software tester how much of the code is being traversed when the test is conducted.

Code coverage is a white-box testing procedure, since the program code needs to be accessible to calculate the code coverage of a software component. During the module testing phase of software development is usually where code coverage is most essential. [4]

3.3.1 Code Coverage Metrics

There are several different types of code coverage that demonstrate how well the code is navigated. Each code coverage metric shows a different way of looking at the manner at which the code can be traversed.

3.3.1.1 Statement coverage (SC)

Statement coverage indicates the percentage of code statements executed. In order to reach 100% statement coverage, every statement, every line of code, must be executed, at least once, during the duration of the test.

$$Statement\ Coveage = \frac{Number\ of\ statements\ executed}{Total\ number\ of\ statements}\ x\ 100\%$$

3.3.1.2 Branch coverage (BC)

Branch coverage indicates the percentage of branches entered. In order to reach 100% branch coverage, all branches, every possibility of IF, IF/ELSE, and SWITHC/CASE statements must be entered, at least once, during the duration of the test.

3.3.1.3 Decision Coverage (DC)

Decision coverage indicates the percentage of decision outcomes occurred out of all the decisions possible. In order to reach 100% decision coverage, all the outcomes of each decision in the code must have occurred during the duration of the test.

$$Decision\ Coverage = \frac{Number\ of\ decision\ outcomes\ exercised}{Total\ number\ of\ decision\ outcomes}\ x\ 100\ \%$$

3.3.1.4 Condition Coverage (CC)

Condition coverage indicates the percentage of condition outcomes out of all the conditions possible. In order to reach 100% condition coverage, all conditional variables, typically Boolean variables, must have occurred with every possible outcome, usually TRUE and FALSE, of that condition.

3.3.1.5 Condition Decision Coverage (C/DC)

Condition decision coverage is a combination of condition coverage and decision coverage. In order to reach 100% condition decision coverage, every possible decision outcome has to have occurred, while the conditions, that made these decisions, must have taken up every possible value.

3.3.1.6 Modified Condition Decision Coverage (MC/DC)

Modified condition decision coverage is the most complex code coverage. In order to reach 100% modified condition decision coverage all the decision outcomes

have to occur. While all the conditions, that made the decisions, must have taken up every possible value. And each variable that was part of such a condition, must have been dominant in influencing the outcome of that condition.

3.4 CUnit Testing Framework

3.4.1 Introduction

CUnit is a testing framework for software programs implemented in C. Its main application is to provide software testers a way to write, administer, and run unit tests. Through numerous flexible user interfaces, it offers programmers a basic testing functionality to test their program code.

CUnit is actually a static library that is linked together with the program code that is to be tested. With the help of its simple framework and broad set of assertions, it allows the testers to construct test structures and to test similar data types. Through the many interfaces CUnit provides, running tests and viewing results can be done dynamically. [6]

3.4.2 Header Files

Functions and data types provide by CUnit can be reached by the user by including the following header files. [6]

- **`CUnit/CUnit.h`:** ASSERT macros for use in test cases, and includes other framework headers.
- **`CUnit/TestDB.h`:** Data type definitions and manipulation functions for the test registry, suites, and tests. Included automatically by CUnit.h.
- **`CUnit/TestRun.h`:** Data type definitions and functions for running tests and retrieving results. Included automatically by CUnit.h.
- **`CUnit/Basic.h`:** Basic interface with non-interactive output to stdout.
- **`CUnit/Automated.h`:** Automated interface with xml output.
- **`CUnit/Console.h`:** Interactive console interface.

3.4.3 Assertions

In software testing, an assertion is a statement on a logical expression that should be true at the location in the code where the assertion is implemented. If at the location of the assertion, the logical expression is false, then the assertion fails. An assertion failing can lead to the program crashing, an assert exception being thrown, or it can be logged in the testing scenarios.

CUnit provides numerous assertions to test different kinds of logical conditions. The framework logs the success and failures of the assertions, and after the test is complete, it is possible to view the results. If an assertion fails, the test continues, except if the assertion is of the "FATAL" type, in which case the test function is aborted and returns instantly. [6]

3.4.3.1 Assert Example:

```
CU_ASSERT_EQUAL(actual, expected);
```

CUnit asserts that the value of actual equals the value of expected. Both actual and expected can be an arbitrary variable (integer, unsigned 8 bit integer, etc.) as long as they are the same type.

3.4.4 Test Registry

The test registry is the collection of the test suites. The user can update the test registry by adding more test suites. The registry is actually data structure, which holds the number of suites and tests that the registry contains, and also pointer to the head of the registered test suites. [6]

```
typedef struct CU_TestRegistry
{
  unsigned int uiNumberOfSuites;
  unsigned int uiNumberOfTests;
  CU_pSuite    pSuite;
} CU_TestRegistry;
```

After testing is complete, the test registry should be cleaned up, by calling the *CU_cleanup_registry* function. The reason is to avoid memory leaks.

3.4.5 Test Suites

A test suite is a collection of test cases that usually examine similar requirements. In order for CUnit to run the test suite, it must be registered in the test registry. The following is the syntax for a collection of test suites. [6]

```
CU_SuiteInfo suites[] = {
  tSuite1,
  tSuite2,
  CU_SUITE_INFO_NULL,
};
```

3.4.6 Test Cases

A test case is a specific test that examines a precise functionality of a software. It provides the expected output of the software in order to compare with the actual output. This is what decides whether or not the test is successful. A test case provides information on whether or not the software component passed the test or not. [6]

4 AUTOSAR Communication Stack

4.1 Introduction

The main responsibility of the communication stack is to assure data transmission between software components. The communication stack consists of a services layer, hardware abstraction layer, and a drivers layer. [1]

4.1.1 Communication Services

The communication services layer is situated between the RTE and the communication hardware abstraction layer. This layer is responsible for the tasks of network management and data exchange, which is accomplished by the Communication Manager (ComM) module and Communication (COM) module, respectively. This layer is entirely independent of the hardware.

4.1.2 Communication Hardware Abstraction

This layer contains the interfaces of the communication bus protocols such as Ethernet, FlexRay, CAN, and LIN. It is dependent of the hardware configuration of the ECU, but independent from the microcontroller employed.

4.1.3 Communication Drivers

This layer contains the drivers of the vehicle communication bus protocols such as CAN, LIN, and FlexRay, and also for onboard communication such as SPI. The drivers are entirely dependent on the microcontroller utilized for the ECU.

4.2 Data Exchange

The data is transmitted between layers of the communication stack by data packets called PDUs (Protocol Data Unit). Each layer has a corresponding protocol for assembling PDUs. Information pertaining to the protocol is included in the PDU, in the form of the Protocol Control Information (PCI) field of the PDU. The PCI consists of information about the source and target layer. The PCI is added to the data packet by the sending layer before transmission, and when the receiving layer obtains the PDU, the PCI is then removed. The Service Data Unit (SDU) is the useful data part of the PDU.

The SDU is the information the upper layer passes to the lower layer. An identifier called the PDU ID is used to identify PDUs throughout the communication stack.

To clarify the process of data transmission the following example is given. A upper layer desires to send data to a lower layer. The data is then assembled into a SDU with a PCI attached. From this point it is called a PDU, and is transmitted to the lower layer. The lower layer receives the PDU, removes the PCI, and obtains the useful data that is contained in the SDU. The above explained process is shown in the figure below. [1]

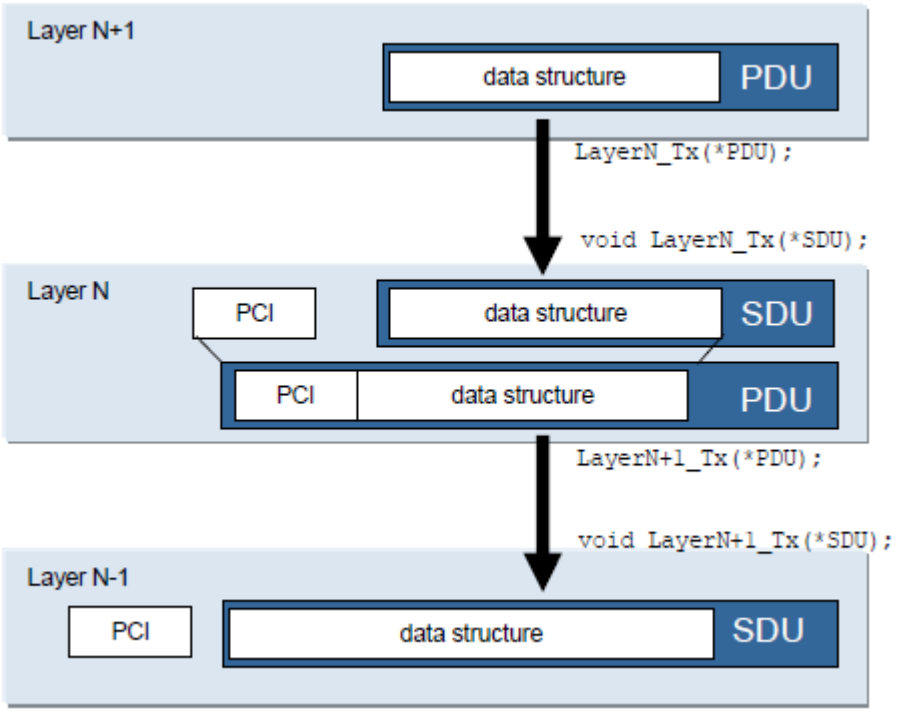

Figure 4.1 Flow of PDUs between Layers [1]

There are three classifications of PDUs, depending on which layer assembled the PDU. The three PDU types are listed below and definition of a signal is explained.

4.2.1 I-PDU

The I-PDU is the PDU of the interaction layer, or the presentation layer of the OSI model. It is assembled in the COM module, and contains signals received from the RTE.

4.2.2 N-PDU

The N-PDU is the PDU of the network layer of the OSI model. It is assembled by the transport protocols, and its size is limited by the maximum size of the data the particular transport protocol can transmit. For example, CAN is limited to 8 bytes.

4.2.3 L-PDU

The L-PDU is the PDU of the data link layer of the OSI model. It is assembled by protocol interfaces of the hardware abstraction layer. In the drivers layer, the L-PDUs are placed on the physical bus for transmission.

4.2.4 Signals

Signals are data packets exchanged between the COM module and the RTE. They are another representation of data used in the automotive industry, it is equal to that of a message in the OSEK standard. [9]

5 I-PDU Multiplexer Module

5.1 Introduction

The IpduM module is an AUTOSAR Basic Software module located in the services layer. It directly communicates with the PDU Router module, and in some cases with the Communication module. As the name suggests, the modules main purpose is handling multiplexed I-PDUs.

When discussing PDU multiplexing, it is meant that a PDU comprises of more than one unique arrangement of its SDU, while using the same PCI. The selector field is part of the SDU, and is responsible for indicating a specific arrangement of the PDU.

IpduM has to combine received I-PDUs from the COM module, and form multiplexed I-PDUs, which are sent to the PDU Router. In this case, the IpduM acts as the sender. When the IpduM module is receiving, it must comprehend the received multiplex I-PDUs and, with the help of the selector field, deliver the corresponding dismantled I-PDUs to the COM module.

The IpduM module is implemented next to the PDU Router in the AUTOSAR Communication Stack. If for some reason, a system does not require multiplexing of PDUs, then the system can be built without the IpduM module. So far, I-PDU multiplexing has only been used for CAN communication, however it is not limited to it. Multiplexing of PDUs are possible on all other communication systems, which can be handled by the PDU Router. Therefore it is technically possible on other bus systems such as FlexRay and LIN. [2]

5.2 Reliance on Other Modules

The IpduM module is relatively isolated compared to other modules in the AUTOSAR Communication Stack. It depends only on total of three other modules. The IpduM module relies on the PDU Router and COM module for communication and relies on the BSW Scheduler for time scheduling. [2]

5.2.1 BSW Scheduler

The BSW Scheduler is generated with the RTE. Its main purpose is to dictate the time intervals for all BSW modules, and correct any problem pertaining to timing inconsistencies. The BSW Scheduler does this by calling the main function of BSW modules. In case of the IpduM module, it periodically calls *IpduM_MainFunction* according to the value configured in *IpduMConfigurationTimeBase*.

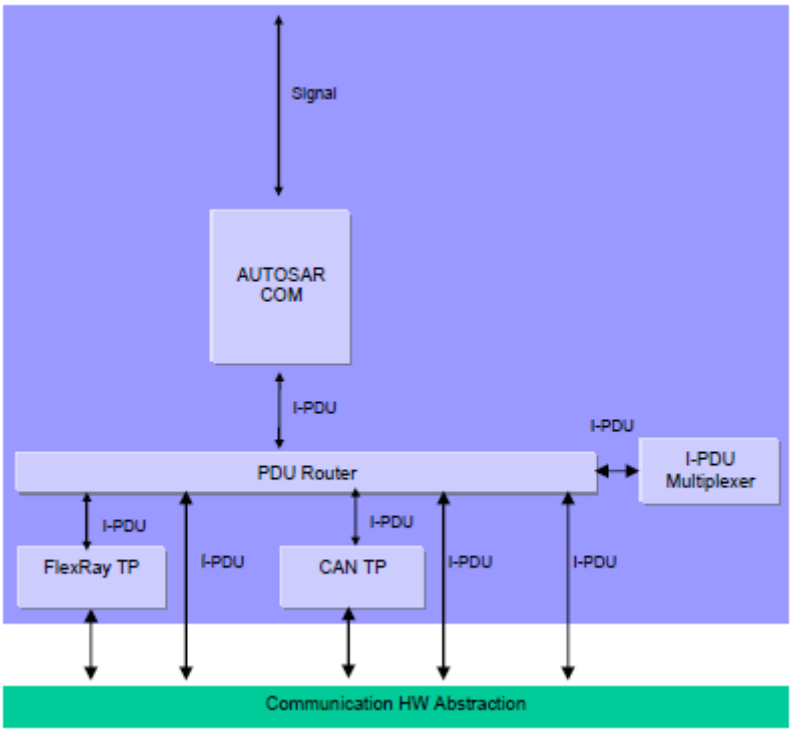

Figure 5.1 Location of I-PDU Multiplexer in AUTOSAR [1]

5.2.2 PDU Router

The main purpose of the PDU Router is to forward I-PDUs between the different modules of the services layer. The PDU Router can also take up a role of a gateway, meaning it must forward an incoming PDU from a lower level module to another lower level module.

The IpduM module needs the following functionalities from the PDU Router. The PDU Router must send an indication to IpduM of incoming multiplexed I-PDUs. It must provide IpduM with a sending interface for outgoing I-PDUs. And must send confirmations of I-PDUs that were successfully transmitted to the IpduM module.

The following functionalities are provided by the IpduM module for the PDU Router. IpduM provides an indication interface for incoming I-PDUs, which are

then de-multiplexed. It also provides a sending interface for I-PDUs, which will be multiplexed. It provides a confirmation interface for transmitted I-PDUs.

The look-up tables and other configurations of the PDU Router, must include I-PDUs which are routed to the IpduM module. These I-PDUs must also belong to multiplexed I-PDUs and must represent a static or a dynamic part.

5.2.3 COM

The COM module communicates with the RTE, the upper layer, through a signal based interface. Communication with lower layers are done through a PDU based interface. This module is responsible for receiving signals from the RTE, and converting the data into I-PDUs in order to forward them to the PDU Router. In the other direction, it is capable of receiving I-PDUs from the PDU Router, convert them into signals, and forward them to the upper layer, the Run Time Environment.

The configurations of both the IpduM module and COM module must be compatible in such a way that for each multiplexed I-PDU, there needs to be other I-PDUs configured in the COM module representing the static part and each arrangement of the dynamic part. Since the COM module configuration contains I-PDUs representing dynamic parts, the IpduM module assumes that it coincides with the correct selector field values.

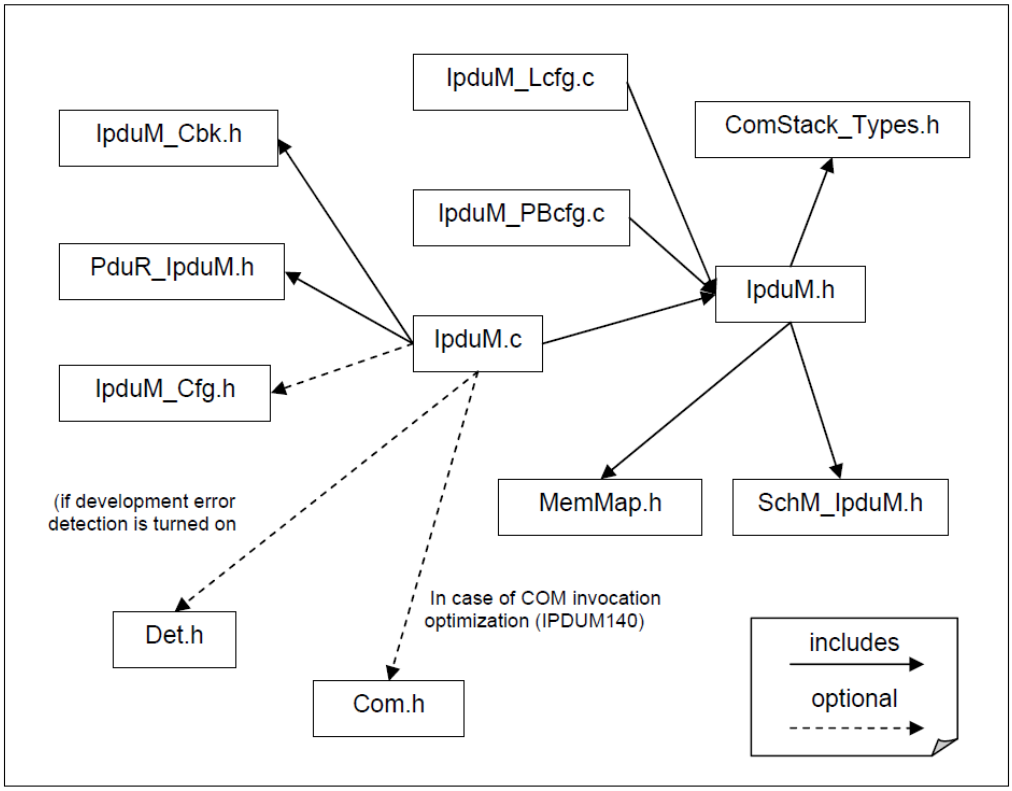

Figure 5.2 Header File Structure [2]

5.3 Multiplexing of I-PDUs

5.3.1 Static and Dynamic Parts

I-PDUs that have been multiplexed contain a static part and a dynamic part. The static part is not essential, and can be omitted. That static part can contain zero or more signals. The dynamic part, however is necessarily, and it must contain the selector field. Therefore, one or more signals are contained in the dynamic part.

The dynamic part of an I-PDU can be thought of as a union in the C language. The value of the selector field decides which layout of the I-PDU is used. Each I-PDU can be configured separately with the static and dynamic parts in different locations. Both static and dynamic parts of the I-PDU can be further partitioned into smaller parts, called segments. Each static or dynamic parts can contain numerous segments. It is important to note however, that there can be several dynamic parts each with different segments, but the area that the segments cover as a whole, must be the identical for each dynamic part of a particular I-PDU. Segments are allowed to cross byte boundaries, and therefore so are parts. [2]

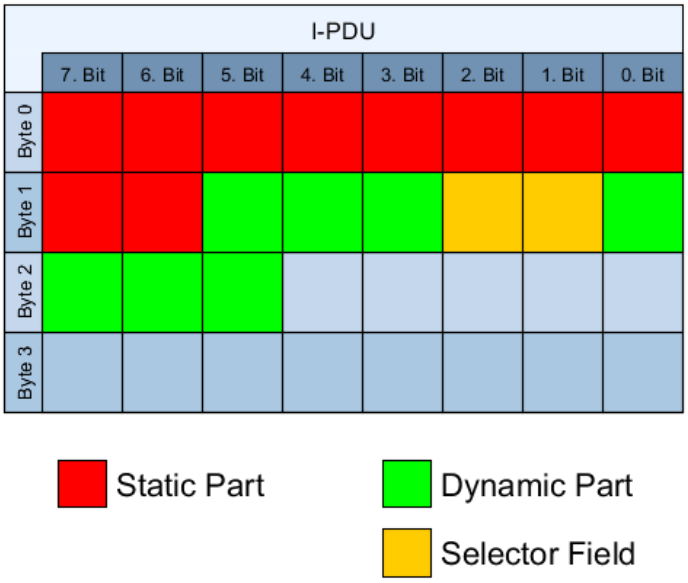

Figure 5.3 Conceptual Layout of an I-PDU

5.3.2 Selector Field

The selector field is located in the dynamic part of an I-PDU, and its size has to be between 1 and 8 adjoining bits long. The selector field must be contained in one single byte, implying that its bits cannot cross into another byte.

The purpose of the selector field is to indicate which dynamic part of the I-PDU is arranged. The size of the selector field decides how many dynamic parts can be configure for a particular I-PDU. For example, if the size of the selector field is 4 bits long, then there can be a maximum of $2^4 = 16$ different dynamic parts configured, with each value of the selector field corresponding to one dynamic part. However, this does not mean that each value of the selector field must correspond to a dynamic part. There can be less dynamic parts than the maximum value of the selector field. [2]

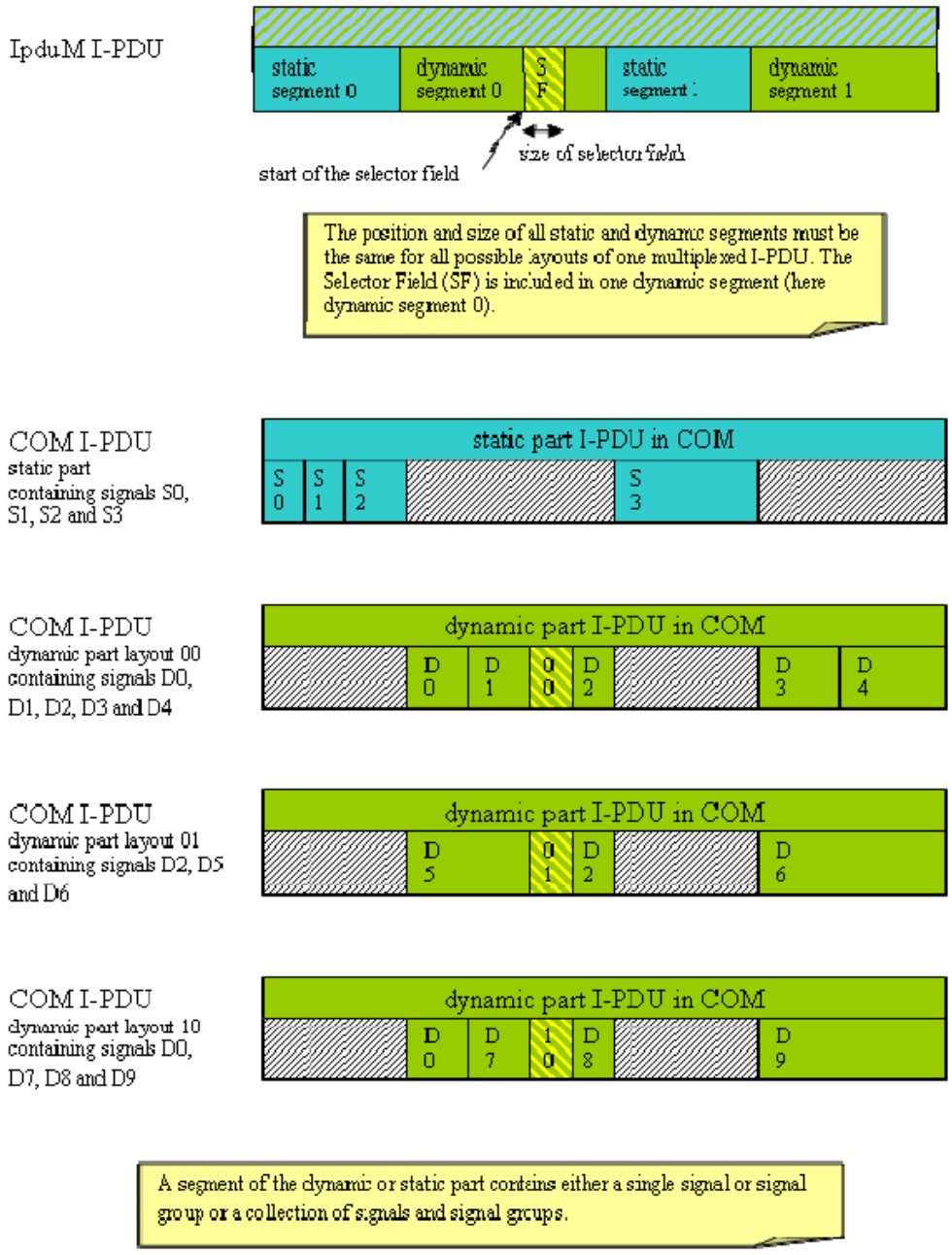

Figure 5.4 Possible layout of a multiplexed I PDU [2]

5.4 Data Types

5.4.1 Type Definitions

Type definitions are used to name data types to make the code more comprehendible. Type definitions are solely for the purpose for helping the programmer understand the code, since after the compiler interprets the type definition, it is substituted by the actual data type. Type definitions are commonly used to name data structures. The following type definitions are used by the IpduM module. [2]

5.4.1.1 PduIdType

PduIdType is an unsigned 8-bit integer (in the case of the IpduM module). It is used to identify a specific PDU.

5.4.1.2 PduLengthType

PduLengthType is an unsigned 8-bit integer (in the case of the IpduM module). It states the length of a specific PDU.

5.4.1.3 Std_ReturnType

Std_ReturnType is an unsigned 8-bit integer. Even though it can take up 256 values, it only utilizes two. The two values are defines, *E_OK* which has a value of 0, and *E_NOT_OK* which has a value of 1. *Std_ReturnType* is used to indicate the return value of functions. *E_OK* indicates the function was executed successfully, and *E_NOT_OK* indicates an unsuccessful function call.

5.4.2 Structures

A structure in C is a collection of different data types. It is frequently used to encapsulate data that relate to one another, and together represent a new entity. The IpduM module utilizes the following structures.

5.4.2.1 PduInfoType

The *PduInfoType* structure represent the PDU. It contains two members, a pointer to a unsigned 8-bit integer, called *SduDataPtr*, and a *PduLengthType*, called *SduLength*. The *SduDataPtr* points to the first byte of the PDU, and the *SduLength* indicates how much of the data afterwards represent the PDU. So technically the *SduDataPtr* is a *SduLength* long array.

5.4.2.2 Std_VersionInfoType

The *Std_VersionInfoType* structure represents the version information of the module. It has five members in total, two unsigned 16-bit integers, *vendorID* and *moduleID*, and three unsigned 8-bit integers, *sw_major_version*, *sw_minor_version*, and *sw_patch_version*. The above mentioned information is irrelevant to the functionality of the module, it just contains data regarding the version of the module.

5.4.2.3 IpduM_ConfigType

The *IpduM_ConfigType* structure contains members that correspond to the parameters in the configuration model. Through this structure all the configured parameters can be reached.

5.5 Functions of the Module

The IpduM module has a total of seven functions, as whole they are known as the Application Programming Interface (API) of the module. Through the API, the module communicates with the rest of the AUTOSAR software architecture. [2]

5.5.1 Initialize

The *IpduM_Init* function initializes the IpduM module. Before this function call the module is considered to be in an uninitialized state, which means it is uncertain what will happen by calling any other function of the module. *IpduM_Init* has one parameter, a pointer to an *IpduM_ConfigType* structure. This structure contains elements corresponding to the configuration parameters selected in the model.

5.5.1.1 Syntax:

```
void IpduM_Init( const IpduM_ConfigType* config );
```

5.5.2 GetVersionInfo

The *IpduM_GetVersionInfo* function is used to obtain the version information of the module. This function can be turned either on or off by a pre compile time configuration parameter called *IpduMVersionInfoApi*. The parameter of the *IpduM_GetVersionInfo* function is a pointer to a *Std_VersionInfoType* structure, where the version information of the module will be stored. This structure contains the module ID and the Vendor ID.

5.5.2.1 Syntax:

```
void IpduM_GetVersionInfo( Std_VersionInfoType* versioninfo );
```

5.5.3 Transmit

The *IpduM_Transmit* function is used to transmit data to the IpduM module by the PDU Router. This function has two parameters, the PDU ID and a pointer to a structure containing the I-PDU, which is to be transmitted. The return value of the

function is a *Std_ReturnType* which has two values, *E_OK* if the transmission was successful, and *E_NOT_OK* if the transmission failed.

5.5.3.1 Syntax:

```
Std_ReturnType IpduM_Transmit( PduIdType PdumTxPduId, const PduInfoType* PduInfoPtr );
```

5.5.4 RxIndication

The *IpduM_RxIndication* function is an indication from the lower layer module, stating that it successfully received the I-PDU. This function contains two parameters, a PDU ID and a pointer to a structure containing the received I-PDU.

5.5.4.1 Syntax:

```
void IpduM_RxIndication( PduIdType RxPduId, PduInfoType* PduInfoPtr );
```

5.5.5 TxConfirmation

The *IpduM_TxConfirmation* function is a confirmation from the lower layer module, to confirm that it transmitted the I-PDU. This function has one parameter, the PDU ID of the transmitted I-PDU.

5.5.5.1 Syntax:

```
void IpduM_TxConfirmation( PduIdType TxPduId );
```

5.5.6 TriggerTransmit

The *IpduM_TriggerT*ransmit function is a request made by the lower layer module, to copy the transmit buffers of the upper layer module. The two parameters of the functions are, the ID of the PDU which is to be transmitted, and a pointer to a buffer where it shall copy the content of the transmit buffers. The return value of this function is a *Std_ReturnType*, which can take up two values. *E_OK* if the PDU was successfully copied, and *E_NOT_OK* if the function was unable to copy the PDU. An unsuccessful copy is usually due to the pointer of the PDU, being a null pointer or its points to invalid data.

5.5.6.1 Syntax:

```
Std_ReturnType IpduM_TriggerTransmit( PduIdType TxPduId, PduInfoType* PduInfoPtr );
```

5.5.7 MainFunction

The *IpduM_MainFunction* is a scheduled function that is called by the BSW Scheduler to synchronous the modules. It is periodically called to signify the time passing. In the IpduM module, this functionality is important because of the *TxConfirmation* timeout that can be configured.

5.5.7.1 Syntax:

```
void IpduM_MainFunction( void );
```

5.6 Initialization

Before the IpduM module can be put into use, it needs to be initialized. This is done by calling the already discussed *IpduM_Init* function. By calling this function, the module is placed into an initialized state, and the internal global variables and buffers of the module are reset and initialized.

The *IpduM_Init* function utilizes the *PduR_IpduMTriggerTransmit* function to acquire the initial values of the internal buffers. Therefore, the COM module must be initialized before the IpduM module. This is the responsibility of the system designer to make sure the modules are initialized in the correct order. [2]

5.7 Transmission

Transmissions of I-PDUs occur in the following way. Inside the COM module, the I-PDUs are separated according to their static and dynamic parts. So each part is considered a distinct I-PDU, with its own PDU ID. When an *IpduM_Transmit* function is called by the PDU Router, the I-PDU corresponding with the PDU ID in the argument of the function call is transmitted to the IpduM module. An *IpduM_TxConfirmation* call can be made, by the PDU Router to confirm the successful transmission of the I-PDU.

After transmit calls have been made, the *IpduM_TriggerTransmit* function takes the transmitted static part and the last transmitted dynamic part and merges it into a new multiplexed I-PDU, with a new unique I-PDU ID. This multiplexed I-PDU is then copied into a buffer specified by the argument of the *IpduM_TriggerTransmit* function. [2]

5.8 Reception

Multiplexed I-PDUs sent by the PDU Router and are received by the IpduM module. When the IpduM module receives the multiplexed I-PDUs, it individually forwards the static part and dynamic parts to their corresponding destinations, by using their distinct PDU IDs.

The multiplexed I-PDU does not need to be divided into different I-PDUs, because the target module receiving the I-PDU will simply just ignore the segments it does not require. For this reason it is simply enough just to forward the same multiplexed I-PDU with different PDU IDs. I-PDUs with a *PduLength* of zero will be silently ignored. [2]

5.9 Development Errors

During development of modules, specific errors are used to indicate unintended use of functions. Detected development errors are reported to the BSW module, called the Development Error Tracer (DET). The configuration parameter, *IpduMDevErrorDetect*, specifies whether or not development errors are to be detected. The IpduM module comprises of the following three development error codes.

5.9.1.1 IPDUM_E_PARAM (0x10)

This development error indicates when a function was called with an invalid parameter. For example, if the *IpduM_Transmit* function was called with a non-existent PDU ID.

5.9.1.2 IPDUM_E_PARAM_POINTER (0x11)

This development error indicates when a function was called with a null pointer. This can occur, for example, when the *IpduM_TriggerTransmit* function is called with the *PduInfoType* pointer pointing to NULL.

5.9.1.3 IPDUM_E_UNINIT (0x20)

This development error indicates when a function was called before initialization of the IpduM module. For example, an *IpduM_Transmit* call is made, before calling *IpduM_Init*.

6 Testing of the Module

This chapter represents the effort I have put into my thesis. All the following sections are about how I created the test environment of the IpduM module. I explain the thought process I used in writing the test cases that examine the IpduM module.

6.1 Prerequisites

Before the formal testing of the module can begin, several prerequisite steps had to be taken. These include understanding the tools used for testing, the requirements that are being tested, and the testing procedure itself. It is important to note that the I-PDU Multiplexer was tested for requirements dictated by 4.0 Release of the AUTOSAR standard.

6.1.1 Tools

6.1.1.1 Eclipse

Eclipse is an open source integrated development environment (IDE), with the ability to develop in multiple programming languages. Eclipse was utilized to develop the testing environment written in the C language. The debugging functionality of Eclipse was also used.

6.1.1.2 In-house Developed Modeling Tool

TKP developed an AUTOSAR modeling and configuring tool specifically for ECU development. It is used to configure the parameters of the module, and to generate configuration header and source files.

For each configuration, the following three files are generated into the *ECUSoftwareIpduMTest.x86.cc* test project.

- **`IpduM_Cfg.h`:** This is the configuration header file containing the pre-compile time configurable parameters. These parameters are defined as macros. For example, *IPDUM_DEV_ERROR_DETECT* is set to *STD_ON*.

- **`IpduM_PBcfg.h`:** This is the configuration header file containing the post-build time configurable parameters. It contains the pointer to the configuration structure and also an array containing pointers to a number of configuration structures.
- **`IpduM_PBCfg.c`:** This is the configuration source file containing numerous structures and data types representing the parameters configured in the model. These data types can be reached using the main configuration pointer, *IpduM_BasicConfigPtr*, or from the configuration pointer array, *IpduM_MultiConfigArray*.

6.1.1.3 MinGW

MinGW, stands for "Minimalist GNU for windows", is an open source software development environment that is predominantly used for making Microsoft Windows applications. MinGW includes the GNU Compiler Collection (GCC), a compiler for various programming languages. The GCC compiler functionality of MinGW was utilized to compile source code written in C. [7]

By accessing the *ECUSoftwareBuilder.all.all* project in the MinGW shell, commands can be given to compile the module and the testing environment together. The following are a list of commands used in the MinGW shell during testing of the module.

- `cd c://company/Eclipse_C_WS/ECUSoftwareBuilder.all.all`

This command accesses the *ECUSoftwareBuilder.all.all* project.

- `build.sh --for-project -e ECUSoftwareIpduMTest.x86.gcc -c ConfBasic -g rebuild`

This command is rebuilds and compiles the IpduM test project with the *ConfBasic* configuration and IpduM module. Instead of rebuild, build can also be used to only affect source code files that were modified since the last build. This saves time during development of the testing environment.

- `../ECUSoftwareIpduMTest.x86.gcc/bin/ConfBasic/ECUSoftwareIpduMTest.x86.gcc.out basic`

This command runs the out file that was compiled by the previous command. By running the out file, the shell displays the test suites and test cases, and indicates which test passed or failed. By adding the keyword, *verbose* after the word basic in the command, all the logs are displayed out on the shell in detail. This was useful, when examining why some test cases failed during their development.

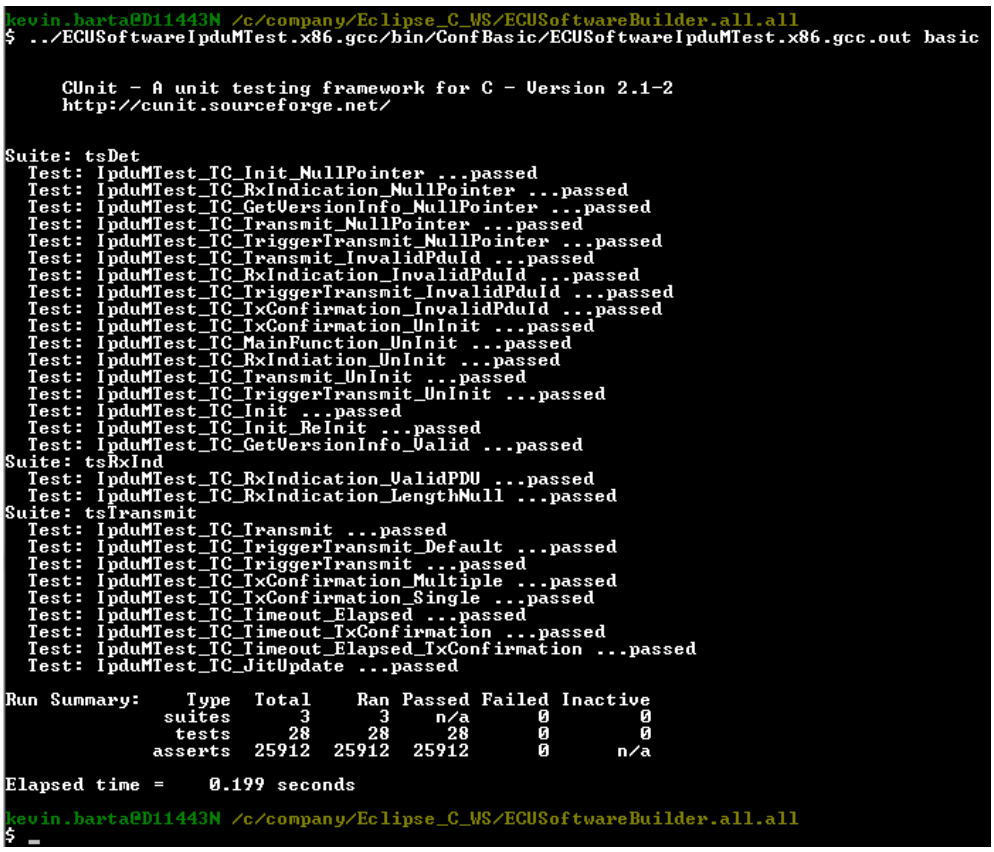

Figure 6.1 MinGW Shell displaying successful test cases

- `build.sh --requirement-analysis -e ECUSoftwareIpduMTest.x86.gcc`

This command runs a requirement analyzing tool. It generates a folder in the test project called *requirement-analysis*, which includes a file that displays the number of requirement tested and the location of the tagged requirement.

6.1.2 Requirements

The IpduM module has a total of 83 requirements. Out of which 5 were proposed by ThyssenKrupp Presta AG, the remaining requirement were proposed by AUTOSAR. However, only 33 requirements were testable. To test them, specific test cases were written to assert the functionality of each testable requirement. After a test case properly satisfies the requirement, it was tagged in the source code with the proper

format, including its requirement ID. Simple requirements are usually tested in one test case. While more complex requirements might need several test cases to fully test the functionality that the requirement proposes.

6.1.3 Testing Procedure

The testing of the IpduM module was done using the black-box method. Only through the interface of the module was the test performed and the knowledge of the inside of the module was unneeded.

AUTOSAR BSW modules are tested as independent components, meaning they are tested without the rest of the AUTOSAR software architecture. The module that is currently undergoing testing is called the System Under Test (SUT). The driver takes the place of upper level modules, which call the module's API functions. Underneath the SUT are stubs, empty modules with only an interface and without any inner architecture. The stubs are representation of modules that interact with the SUT, and their purpose is to log testing information when their API functions are called.

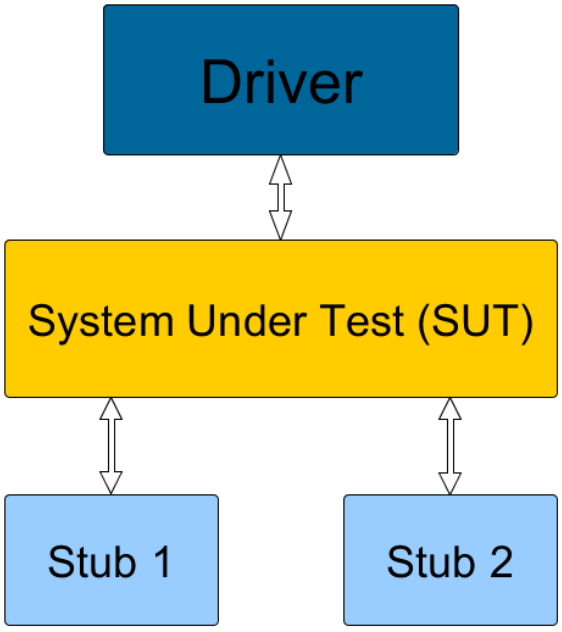

Figure 6.2 Conceptual Diagram of the Testing Environment

Each stub receives information it needs to log. These stubs logs hold the data that the IpduM module used to interact with the stubs. The stub logs reflect the actual behavior of the IpduM module that was implemented. During test cases, the APIs of the module are called, and data is written into other stub logs, called the expected stub logs.

By calling APIs, the actual stub logs are modified. The data written in the expected stub logs are according to the behavior of the module dictated by the requirements in the specification AUTOSAR releases for the module. Therefore the actual and expected stub logs are modified entirely independent of each other.

The actual stub logs represent the actual behavior of the IpduM module, while the expected stub logs represent the expected behavior according to the requirements. These two array of values are compared inside test cases to check if the IpduM module behaves according to the requirements. If the values of the stub logs are equal, then the module behaves properly and the testing environment is set up sufficiently. If there is a difference, then either the test is wrong, or the module does not behave according to the specifications.

The goal is to have the expected and actual stub logs be the same, but only if the IpduM module is implemented properly. The IpduM module is deemed fully tested if the following three conditions are satisfied. The test cases check all the requirements. During each test case the expected and actual stubs are equal. The branch coverage of the module is 100%.

6.2 Testing Environment

The testing of the module is done with the help of the CUnit framework, and it must be set up correctly before testing can begin. The following sections indicate important steps to be taken before test cases can be implemented.

6.2.1 Framework Setup

The entire testing environment is ran by the *main.c* source file. The *main* function is the entry point of the program. It initializes the test registry, adds the utilized test suites, and cleans up the registry after the tests are completed.

The test suites are defined in the *testsuites.h* header file as external global variables. The test suites are *CU_TestInfo* data structures that contain the signatures of their test cases. The test suites are registered using the *registerTestSuites* function in the *main* file.

6.2.2 Test Utilities

The *testhelper.h* header file contains the values that can be determined before run-time. These values are defined as macros and are used for specific reasons such as setting maximum and default values for buffers and IDs, and also to establish error codes for the API functions. The IDs and error codes are specified by AUTOSAR and their values do not change for a specific module. The maximum and default values for buffers and IDs are used in multiple locations throughout the code. If for some reason they need to be changed, only the value of the macro has to be replaced with a different value.

When writing test cases, sometimes it occurs that the same line of code is written down multiple times. It is at this point, when the tester realizes that such line of code should be placed into a function and called whenever the functionality is needed. A collection of functions frequently used throughout the code are found in the *testhelper.c* file to assist the test cases. These functions are explained in detail below.

6.2.2.1 TestHelper_Init

In the test cases, this function is called before calling *IpduM_Init,* to guarantee that the IpduM module is in an uninitialized state. This is done by setting the global variable *IpduM_Initialized* to *FALSE*. The unsigned 8-bit integer parameter, *defaultData* passes its value to the global variable *defaultUsedData*, which is the default value of the used areas of an I-PDU. The *ResetStubLogs_All* is called to reset the expected and actual stub logs.

6.2.2.2 TestHelper_CheckArrays

This function compares the value of two arrays to check whether they are equivalent or not. First the length of the two arrays are compared, and if they are the same, the function proceeds to compare the elements of the array. If the lengths are different, then the *CU_ASSERT_EQUAL* is called on the lengths, and since they are not equal, the assert fails.

The corresponding elements of the two arrays are then compared and if there is any discrepancy, a Boolean variable, called *correct*, is set to *FALSE*. At the end of the function, an assertion is made that the *correct* variable stayed *TRUE.* If *correct* is set to *FALSE*, then the assertion fails.

6.2.2.3 TestHelper_InitPduRTriggerTransmit

During initialization, the *IpduM_Init* function utilizes the *PduR_IpduMTriggerTransmit* function to acquire the initial values of the internal buffers. This means the actual logs of the PDU Router stub have been augmented. So the corresponding data must be added to the expected logs as well, to reflect the changed in the actual logs caused by the *PduR_IpduMTriggerTransmit* function call.

Since this needs to be done before every initialization in each test case, it was reasonable to create a function that fills the expected logs of the PDU Router corresponding to the first *PduR_IpduMTriggerTransmit*.

6.2.2.4 TestHelper_InitComTriggerTransmit

This function does the same task as the *TestHelper_InitPduRTriggerTransmit* function, expect it fills the expected logs of the COM stub, instead of the PDU Router stub. It depends on whether the *IpduMRxDirectComInvocation* is turned on in the configuration, in which case the *TestHelper_InitComTriggerTransmit* is called instead of *TestHelper_InitPduRTriggerTransmit*.

6.2.2.5 TestHelper_HammingWeight

This function calculates the hamming weight of the byte that was given to the function as a parameter, *aNumber*. The thought process behind calculating the hamming weight is that a mask is used to determine the value at each bit, and the number of 1 bits are added up to get the hamming weight of *aNumber*. The hamming weight of the *aNumber* is the return value of the function.

6.2.2.6 TestHelper_BitPosition

This function calculates the position of the first 1 bit starting from the least significant bit. The parameter, *aNumber,* is shifted to the right until a 1 bit is reached. A mask determines whether or not the bit is equal to 1 or not. The number of shifts is tallied and it is equal to the bit position of the first 1 bit, viewing the byte from the least significant bit. This value is the return value of the function.

6.2.2.7 TestHelper_ResetPdu

This function overwrites the data of the PDU with the *defaultByte* given as a parameter. The parameter *pdu* is given as a pointer to the PDU that data needs to be modified. The value of the *defaultByte* is copied to each byte of the *pdu*.

6.2.2.8 TestHelper_FillBits

This function fills up a PDU with 1 bits according to the length and position of the segment specified, while taking endianness into consideration. The PDU that is altered is then passed to the function as a pointer to a *PduInfoType,* called *pdu*. The *segment* parameter is an *IpduM_SegmentType* structure containing the starting bit position of the segment and the segment length. A Boolean variable, *bigEndian* indicates whether the PDU should be filled up regarding the big endian byte order (*bigEndian* set to *TRUE*) or the little endian byte order (*bigEndian* set to *FALSE*).

The *segment* can cross byte borders, which is why the endianness is necessary. If big endian is used, then the remaining data is written on the lower index bytes when the byte borders are traversed. If little endian is used, then the upper index bytes are written over if byte borders are crossed.

This function is additive, meaning 1 bits are only added to the *pdu*, it never overwrites 1 bits to 0 bits. This functionality is necessary because several segments can be used to create a mask out of the *pdu*, and the previous modifications should not be overwritten.

6.2.2.9 TestHelper_ChangeData

This function alters the data of the *pdu*, passed as a pointer to a *PduInfoType*. It shifts the even indexed bytes to the left, and the odd indexed bytes to the right.

6.2.2.10 TestHelper_InitBufferLogs

This function initializes the transmit logs of the PDU Router stub. The parameter of this functions is an unsigned 8-bit integer, called *configIdx*, which is used to reach the relevant *IpduMTxPathway* from the configuration model.

With the help of a mask, which is created inside the function, the transmit logs are initialized with the value of *defaultUsedData*, wherever segments exist in the PDU

in the *IpduMTxPathway*. The other areas, where there are no segments, are filled with the value of *IPduUnusedAreasDefault*.

6.2.2.11 TestHelper_CreateInitBuffer

This function initializes the *outPduPntr*, a pointer to a *PduInfoType*, the same way as the previously discussed *TestHelper_InitBufferLogs* function initializes the transmit logs.

6.2.3 Configuration

Using the in-house developed modeling tool, the IpduM module can be configured with different parameters. To thoroughly test the module at least three configurations had to be constructed; *ConfBasic*, *ConfNoDet*, and *ConfNoStatic*.

The *ConfBasic* configuration has all possible parameters configured, including *IpduMDevErrorDetect*, *IpduMStaticPartExists*, *IpduMRxDirectComInvocation*, and *IpduMVersionInfoApi*. This configuration model has numerous *IpduMTxPathways* and *IpduMRxPathways* configured, each with a different combination and values configured for their parameters. The reason for this is to run the test cases through numerous variations of the configurations, in order to comprehensively test the IpduM module. Using the *ConfBasic* configuration all the test cases were conducted.

The ConfNoDet configuration has development error detection turned off. Therefore the test cases examining development errors will be omitted, when running the tests using this configuration.

The ConfNoStatic configuration indicates that no static parts of the PDUs are configured inside the pathways of this configuration.

The static part and dynamic parts of PDUs are configured using this in-house developed modeling tool. The segments can be arbitrarily chosen using this tool, however the specification dictates some constraints, which must be followed in order to effectively test the module. The specification gives more details of what values and parameters are allowed to be configured.

6.3 Stubs

The IpduM module requires communication with two other modules. These two modules are represented as stubs during the testing process. During the development

process, another module, the Development Error Tracer (DET), exchanges data with the IpduM module, therefore it is implemented as a stub as well. Another stub is utilized to ensure mutual exclusion is not infringed upon. The stubs log the information received through the APIs of the IpduM module. There are two commands concerning the stub logs, reset and compare.

The *ResetStubLogs_All* function initializes all the stub logs, both the actual and expected logs. The *CompareStubLogs_All* function compares all the corresponding actual and expected stub logs in order to check whether the tests were successful or not.

6.3.1 Stub Structures

The following stub structures are used by various logs in the testing process of the IpduM module. They are explained here collectively and mentioned later in the sections concerning the stubs that use them.

6.3.1.1 RxIndication

The *RxIndication* log contains the PDU ID of the last two *RxIndication* calls, the SDU data of all the past calls, the entire length of the SDU data, and number of times an *RxIndication* function occurred.

6.3.1.2 TriggerTransmit

The *TriggerTransmit* log contains the PDU ID of all the *TriggerTransmit* calls that occurred, the entire length of the SDUs that were transmitted, and the number of times a *TriggerTransmit* function was called.

6.3.1.3 Transmit

The *Transmit* log contains the PDU ID and data of SDUs of all the past transmit calls, the entire length of the SDUs, and the number of times *Transmit* was called.

6.3.2 COM

The COM module communicates with the IpduM module using the following three APIs: *Com_RxIndication*, *Com_TriggerTransmit*, and *Com_TxConfirmation*. The corresponding logs to these functions are structures embedded in the *COM_InterfaceStubLog* structure.

The *Com_TriggerTransmit* function logs the information pertaining to the *TriggerTransmit* stub log, and also fills the SDU of the PDU pointed to by *pduInfoPtr* with the *defaultUsedData* values. This is because *TriggerTransmit* calls, in general, copy the content of the transmit buffers into the PDU passed as a pointer, and in this case the initial contents of the transmit buffers are set to *defaultUsedData*, in the testing environment.

Com_RxIndication and *Com_TxConfirmation* function simply log the data corresponding to their respective structures.

6.3.3 PduR

The PDU Router module communicates with the IpduM module using the following four API functions: *PduR_RxIndication, PduR_TriggerTransmit, PduR_IpduMTransmit,* and *PduR_TxConfirmation*.

All the above mentioned functions, log the information corresponding to their respective stub structures. The *PduR_IpduMTransmit* function also copies the content of the transmit buffers into the PDU pointed to by the *pduInfoPtr*. The reason was explained in the previous section regarding the COM module.

6.3.4 Det

The stub representing the Development Error Tracer logs development errors, if *IpduMDevErrorDetect* parameter is set in the configuration model. The *ApiId* and the *ErrorId* of all the development errors and number of errors reported are saved in *DET_ErrorReporterStubLog*.

6.3.5 Mutual Exclusion

There is a stub, called *SCHM_StubLog,* which monitors the mutual exclusion requirement. This stub contains only one element, an unsigned 8 bit integer, called *CriticalSection,* which represents if a critical section has been entered or not. The *SchM_Enter_IpduM_EA* function increments *CriticalSection,* indicating that a crictical section was entered. Before entering a critical section, the value of *CriticalSection* had to be zero, because the processes cannot enter their respective critical section at the same time. *SchM_Exit_IpduM_EA* function decrements *CriticalSection*, indicating the program leaving the critical section. If mutual exclusion is not fulfilled, then test fails.

Mutual exclusion is a vital requirement pertaining to multi-thread all software programs.

6.4 Test Suites

The test suites are a collection of test cases examining similar functionalities of the module. The test cases are categorized into three different test suites; *tsDet.c*, *tsRxInd.c*, and *tsTransmit*. These test suites are declared as external *CU_TestInfo* structures in the *testsuite.h* header file, and registered using the *registerTestSuites* function in the *main.c* source file.

6.4.1 Development Test Suite

The *tsDet.c* test suite examines the module for development errors and initialization. The detection of development errors is either turned on or off, by a macro configured during pre-compile time. If *IpduMDevErrorDetect* is set to *FALSE*, then test suite will not run any test cases concerning development errors, it will only run test cases regarding the initialization of the module. If *IpduMDevErrorDetect* is set to *TRUE*, then development errors are detected and reported while running the corresponding test cases.

6.4.1.1 Harness Functions

Harness functions are support functions that are called numerous times during the test suite with different valued parameters. Harness functions cause the test suite to be more compact and comprehendible. There is a harness function for the following IpduM module functions: *IpduM_RxIndication*, *IpduM_TxConfirmation*, *IpduM_TriggerTransmit*, and *IpduM_Transmit*. These harness functions have the same parameters as their corresponding function with an addition of a *uint8* parameter, called *errorCode*.

The *errorCode* parameter represents the development errors, which takes up the hexadecimal value of 0x10, 0x11, and 0x20, each corresponding to a development error. The harnesses were created for following reason. Since the API functions must be tested for all three development errors, the harness is called throughout the test suite three times, performing the same function except with different error codes.

The role of the harness is to call the corresponding API function with the given parameters and to log the *ApiId* and *ErrorId* in the DET stub logs, in the case of error detection.

Inside the harness functions, the *IPDUM027* and *IPDUM028* requirements are tested.

- **IPDUM027:** This requirement states if *IpduMDevErrorDetect* is set to *FALSE,* then development errors shall not be logged. This requirement is fulfilled because inside the harness functions are pre-compile switch directives that decides whether or not the development errors should be logged.
- **IPDUM028:** This requirement is the opposite of the above stated requirement. It states that the development errors should logged if the *IpduMDevErrorDetect* is set to TRUE. This requirement is fulfilled because of the same precompile switch mentioned above.

6.4.1.2 Invalid PDU ID Test Cases

Invalid PDU ID falls under the invalid parameter development error. These test cases examine the effects of calling the function with a PDU ID, which is not configured in the module. The following API functions are tested against invalid PDU ID: *IpduM_RxIndication, IpduM_TxConfirmation*, *IpduM_TriggerTransmit*, and *IpduM_Transmit*.

Inside each test case their corresponding harness functions are called with the *pduId* parameter being equal to the *TEST_IPDUM_PDUID_INVALID* macro and the errorCode parameter being equal to the *TEST_IPDUM_E_PARAM* macro.

Both the above mentioned macros are defined in the *testhelper.h* header file. The value of *TEST_IPDUM_PDUID_INVALID* is defined as 255, because no PDU ID is configured to this value. The hexadecimal value of *TEST_IPDUM_E_PARAM* is defined as 0x10, because the specification defines this as the error code for an API called with an invalid parameter.

The invalid PDU ID test cases are rather simple and only test the *IPDUM026* requirement.

- **IPDUM026:** This requirement states that if an API service is called with an invalid parameter, then the development error code with a hexadecimal value of 0x10 shall be logged. This is fulfilled because the value of the TEST_IPDUM_E_PARAM macro is defined to be 0x10.

6.4.1.3 Null Pointer Test Cases

Null pointer test cases call the API functions with a null pointer as one of its parameters. The following five functions have a pointer as a parameter: *IpduM_RxIndication*, *IpduM_TriggerTransmit*, *IpduM_Transmit*, *IpduM_Init*, and *IpduM_GetVersionInfo*.

IpduM_RxIndication, *IpduM_TriggerTransmit*, and *IpduM_Transmit* all have corresponding harness functions. So in order to properly log the development error, the harness has to be called with the PduInfoType pointer parameter equal to zero, the null pointer, and the error code has to be equal to *TEST_IPDUM_E_PARAM_POINTER*. The remaining parameter, *PduIdType*, is set equal to a correct value obtained from the configuration model using *IpduM_BasicConfigPtr*.

IpduM_Init and *IpduM_GetVersionInfo* both only have one parameter, which is a pointer to an *IpduM_ConfigType* and a pointer to *Std_VersionInfoType*, respectively. In their corresponding null pointer test cases, these pointers are set to equal zero, to signify a null pointer. If the detection of development errors is enabled, their respective *ApiId* and the error code for null pointer, hexadecimal value of 0x11, are both logged in the DET stub log.

Null Pointer test cases test the *IPDUM162* requirement.

- **IPDUM162:** This requirement states that if an API function is called with a null pointer, then the development error code with the hexadecimal value of 0x11 should be logged. This requirement is fulfilled because the value of the *TEST_IPDUM_E_PARAM_POINTER* macro is defined to be 0x11.

6.4.1.4 UnInit Test Cases

UnInit test cases call the API functions before the IpduM module has been initialized by the *IpduM_Init* function. These test cases are necessary to check if the development error, pertaining to calling an API function without module initialization,

is properly logged in the DET stub logs. The following API functions are tested for this; *IpduM_RxIndication*, *IpduM_Transmit, IpduM_TxConfirmation IpduM_Mainfunction*, and *IpduM_TxConfirmation*.

IpduM_Init and *IpduM_GetVersionInfo* are not tested because these two functions are allowed be called before the module is in an initialized state.

The API functions that are tested here all have harness function except one, *IpduM_Mainfunction*. The API functions with harnesses are tested the following way. Inside the test cases, no *IpduM_Init* function call is made, so the module in not initialized. Then the harness functions are called with correct parameters obtained from the configuration model, and while the *errorCode* parameter set equal to *TEST_IPDUM_E_UNINIT*.

The test case examining the *IpduM_Mainfunction* is set up the same way. Without the initialization of the module, the *IpduM_Mainfunction* is called and its *ApiId* and the value of *TEST_IPDUM_E_UNINIT*, hexadecimal 0x20, is logged in the DET stub logs.

The *Uninit* test cases test the following requirements: *IPDUM084*, *IPDUM153*, and *INTERNALREQUIREMENT01*.

- **IPDUM084:** This requirement states that the behavior of the IpduM module is unspecified before a proper *IpduM_Init* function call is made. This requirement is fulfilled because since the API functions were called without proper module initialization, the API did not perform the correct task, therefore the behavior of the module cannot be determined.
- **IPDUM153:** This requirement states that the hexadecimal value of the error code pertaining to calling API services before initialization should be 0x20. This is fulfilled because the value of the *TEST_IPDUM_E_UNINIT* macro is defined to be 0x20.
- **INTERNALREQUIREMENT01:** This requirement states that in an uninitialized state the *IpduM* module will not execute any of its API functions, with the exception of *IpduM_Init* and *IpduM_GetVersionInfo*. This requirement is fulfilled because all the other functions were not executed when the module was not initialized, only their corresponding error codes were logged.

6.4.1.5 Initialization Test Cases

There are two test cases specifically dealing with the initialization of the module, *IpduMTest_TC_Init* and *IpduMTest_TC_Init_ReInit*. *IpduMTest_TC_Init* simply calls the *IpduM_Init* function with the correct configuration pointer, *IpduM_BasicConfigPtr*, to place the module in an initialized state. The *INTERNALREQUIREMENT03* requirement is explicitly fulfilled in this test case.

- **INTERNALREQUIREMENT03:** This requirement states that by the result of a successful *IpduM_Init* call, the module is placed in an initialized state from an uninitialized state.

The *IpduMTest_TC_Init_ReInit* test case asserts that if an *IpduM_Init* function call is made, while the IpduM module is already in an initialized state, it shall be ignored and a development error is logged with the error code value of *TEST_IPDUM_E_INVALID_REINIT*. This test case was made to specifically fulfill the *INTERNALREQUIREMENT02* requirement.

- **INTERNALREQUIREMENT02:** This requirement states that if the IpduM module is in an initialized state, any *IpduM_Init* function calls shall be ignored.

6.4.1.6 GetVersionInfo Test Case

This test case asserts the proper behavior of the *IpduM_GetVersionInfo* function. The *IpduM_GetVersionInfo* function must acquire the correct version information of the IpduM module. This information is copied into the corresponding elements of the *Std_VersionInfoType* structure, called *version*. This structure is passed by reference to the *IpduM_GetVersionInfo* function.

This test case fulfills two requirements, *IPDUM038* and *INTERNALREQUIREMENT01*. The *INTERNALREQUIREMENT01* requirement was already discussed earlier, however this test case fulfills the requirement by giving a sufficient reason why it is fulfilled.

- **INTERNALREQUIREMENT01:** This requirement also states that the *IpduM_GetVersionInfo* function can be called even if the module is in an uninitialized state.

- **IPDUM038:** This requirement states that the *IpduM_GetVersionInfo* shall return the version information of the module, including Module ID and Vendor ID.

6.4.2 Reception Test Suite

Only the *IpduM_RxIndication* function deals with the reception side of the IpduM module, therefore an entire test suite is dedicated to this function. The *tsRxIndication.c* test suite examines the functionality of the *IpduM_RxIndication* function in two test cases, with the help of one harness function.

6.4.2.1 HarnessRxIndication

The harness function receives two parameters, an unsigned 8-bit integer, *configIdx* and a pointer to a *PduInfoType* structure, *pduPntr*. The configIdx is an index used to select the correct *IpduMRxPathway* from the configuration model. The *pduPntr* points to the received PDU which must be separately forwarded by using the static and dynamic part ID.

The harness takes the *IPDUM_RX_DIRECT_COM_INVOCATION* macro into account, which decides whether the COM or PDU Router stub logs should be modified. If *IPDUM_RX_DIRECT_COM_INVOCATION* is set to TRUE, then the COM module sent the PDU directly, so the COM stub logs need to be filled out. If however, the macro is set to FALSE, then the PDU was sent by the PDU Router, and the PduR stub logs need to be augmented.

The harness takes the length of the PDU into consideration, because if it is set to zero, then the PDU must be disregarded. This means none of the stub logs need to be changed. If the PDU has a static part configured, then the stub logs are filled with information regarding to the static part, such as the *OutGoingStaticPduId* of the PDU.

The next step is finding out the value of the selector field. By using information provided by the configuration model, the value of the selector field can be calculated using the *SelectorFieldMask* and *SelectorFieldBytePosition* parameters. If the value of the selector field corresponds to any of the dynamic parts configured, then the *OutGoingDynamicPduId* is saved in the appropriate stub log.

At the very end of the harness, the *IpduM_RxIndication* function is called with the *RxHandleId* of the current *IpduMRxPathway*, and the PDU pointed to by the *pduPntr*.

The purpose of this harness is to calculate the value of the selector field, and to forward the applicable static and dynamic parts of the PDU.

6.4.2.2 RxIndication Test Cases

There are two test cases examining the reception side of the module. *IpduMTest_TC_RxIndication_ValidPDU* test case checks the behavior of the *IpduM_RxIndication* function if it is correctly called with a valid PDU information. This test is conducted for each *IpduMRxPathway* configured in the configuration model. To thoroughly test the *IpduM_RxIndication* function each possible variation of the selector field must be set, in order to test if the correct dynamic parts are forwarded to their respective destination.

The test cases obtains the length of the selector field, and calculates the maximum value the selector field can contain, to use as the end condition for a *for* cycle. The *for* cycle is then used to call the harness of the *IpduM_RxIndication* function with all the possible values of the selector field.

IpduMTest_TC_RxIndication_LengthNull test case asserts that the *IpduM_RxIndication* function properly ignores a PDU which is configured with a *PduLength* of zero. A *PduInfoType* structure, called *pduNull*, is declared with a *PduLength* of zero. The *pduNull* is then passed by reference to the harness of the *IpduM_RxIndication* function. As discussed before, the harness checks whether or not the length of the PDU is zero, and if it is, the stub logs are not modified. The *IpduM_RxIndication* function is still called, but the module will silently ignore it. Since the tsRxIndication test suite only examines the *IpduM_RxIndication* function using two similar test cases, the requirements they fulfilled are listed together.

- **IPDUM041:** This requirement states that if the *IpduM_RxIndication* function is called with a parameter of a PDU containing a static part, then the PDU is forwarded with the help of the static part's ID. This requirement is fulfilled by the harness where it examines the configuration of a static part. If there is a static part configured, then the stub log is filled out with the *OutGoingStaticPduId* of the PDU.

- **IPDUM042:** This requirement states that by the result of calling the *IpduM_RxIndication* function, the dynamic part's ID of the PDU is calculated using the selector field, so the PDU can be forwarded to the dynamic part's destination. This requirement is also fulfilled in the harness when the value of the selector field is calculated, in order to find the correct dynamic part. The *OutGoingDynamicPduId* of the dynamic part is then logged in the appropriate stub log.
- **IPDUM098:** This requirement states that the IpduM module does not set or modify the value of the selector field. This requirement is implicitly fulfilled, since the module does not overwrite the selector field. This can be concluded because the test cases ran successfully.
- **IPDUM140:** This requirement states that the COM module can directly communicate with the IpduM module without the need of the PDU Router. This functionality is turned on by the pre-compile time macro called *IPDUM_RX_DIRECT_COM_INVOCATION*. If it is set to *TRUE*, then the COM module directly communicates with the IpduM module. If it is set to *FALSE*, the communication is directed via the PDU Router.

6.4.3 Transmission Test Suite

The ts*Transmit.c* is the largest and most complex test suite used in the testing of the IpduM module. The complexity is due to the extensive testing that was needed to test precise functionalities such as Just-In-Time updates and *TxConfirmation* timeouts. This test suite deals with the transmission side of the IpduM module, and encompasses the functions; *IpduM_TriggerTransmit*, *IpduM_TxConfirmation*, and *IpduM_Transmit*.

6.4.3.1 Logging Functions

Two functions were implemented specifically to log information, due to the result of Just-In-Time updates and *TxConfirmation* timeouts.

IpduMTest_Log_JIT_TriggerTransmit function logs the *PduIdType* parameter called *handleId* in the trigger transmit logs of the either the PDU Router or the COM module, depending on how the *IPDUM_RX_DIRECT_COM_INVOCATION* macro is configured.

IpduMTest_Log_TxConfirmation function logs the *StaticHandleId* and *DynamicHandleId*, if *StaticConfirmation* and *DynamicConfirmation* are set to *TRUE*, respectively. The *DynamicHandleId* is chosen with the help of the *dynIdx* parameter, which selects the relevant dynamic part from the *DynamicPartConfigArray*.

6.4.3.2 Harness Functions

Two harness functions are used for the transmission of PDUs. There is a separate harness for the *IpduM_Transmit* function regarding the static part and the dynamic part. The reason for creating a distinct harness for both the static and dynamic part is to limit the size of the harness functions and to be able to make a single transmit call.

Both the harness for the static part and dynamic part are rather complex, and without the actual source code, it is difficult to explain their algorithm. So instead, the main purpose and functionality of the harness is explained only.

IpduMTest_HarnessTransmit_StaticPart transmits the static part of the PDU, and fills in the appropriate stub logs. It also takes Just-In-Time updates into consideration. Through the arguments of the harness function, the function receives information regarding the index of the last transmitted dynamic part, as a *uint8* named *lastTransmittedDynamicIdx*. This information is needed to correctly generate the transmitted PDU, if *JitUpdate* is configured. It is also important to know if a there even was a dynamic part transmitted prior to this harness call. This is indicated by the Boolean variable called *dynamicTransmittedBefore*. If there was a dynamic part transmitted prior, then PDU in transmission must represent the dynamic segments as well. The *outPduPntr* PDU pointer is filled out the exact same way as the stub logs, because there are test cases which will need to compare this value without the value returned by the *IpduM_TriggerTransmit* function. If *StaticTrigger* is configured, then the stub logs are filled, and either way the IpduM_Transmit is called with the relevant *StaticHandleId*. If the *JitUpdate* of the dynamic part is configured, then the *IpduMTest_Log_JIT_TriggerTransmit* function is called with the handle ID of the relevant dynamic part. The return value of the harness is a Boolean variable indicating if *StaticTrigger* is configured.

IpduMTest_HarnessTransmit_DynamicPart works in a similar way as the harness for the dynamic part. It transmits the dynamic part of the PDU, by taking the

Just-In-Time updates into account. The harness receives the index of the dynamic part, *dynPartIdx* and whether or not at static part was transmitted before, *staticTransmittedBefore*. If *staticTransmittedBefore* is *TRUE* and the *JitUpdate* is configured, then the static part is also represented in the transmitted PDU. The PDU pointed to by the *outPduPntr* pointer is filled out the same way as the data inside the transmit stub logs. If *DynamicTrigger* is configured the stubs are filled out, and if the *JitUpdate* of the static part is also configured the *IpduMTest_Log_JIT_TriggerTransmit* function is called. *IpduM_Transmit* call is made with the *DynamicHandleId* of the relevant dynamic part. The return value of this harness is the value of the *DynamicTrigger*.

6.4.3.3 Transmit Test Case

This case asserts if the *IpduM_Transmit* function is called properly and the transmission request is accepted. This test case is conducted for each *IpduMTxPathway* and with different data inside the PDU. The harness functions for the static part and all the dynamic parts configured are called for each individual PDU. The following requirements are fulfilled by this test case.

- **IPDUM017:** This requirement states that the *IpduM_Transmit* function must assemble a PDU, with the relevant static and dynamic parts, and transmit it depending on the trigger conditions configured. This requirement is fulfilled inside the harness functions, because the stub logs are filled out if the trigger conditions are met. This mean that the stub logs are filled out only if the transmission is accepted.
- **IPDUM021:** This requirement states that the transmission request is accepted according to the following trigger conditions: static part, dynamic part, static and dynamic part, and none. This requirement is fulfilled in the harness by using subsequent if conditions to fill out the logs, according to the value of the *StaticTrigger* and the *DynamicTrigger* variables.

6.4.3.4 TriggerTransmit Test Cases

There are two test concerning the *IpduM_TriggerTransmit* function. One test case pertains to calling the *IpduM_TriggerTransmit* function without previous *IpduM_Transmit* function calls. This means the default values of the transmit buffers

are received by the *IpduM_TriggerTransmit* function. The other test case performs *IpduM_Transmit* function calls before the *IpduM_TriggerTransmit* function call, while taking the Just-In-Time update into account.

IpduMTest_TC_TriggerTransmit_Default test case asserts that the *IpduM_TriggerTransmit* function properly copies the content of the I-PDU transmit buffer into the buffer pointed to by the *PduInfoPtr*. Since *IpduM_Transmit* was not called prior to the *IpduM_TriggerTransmit*, the transmit buffers contain the default values. The test is conducted for each *IpduMTxPathway*. The *IpduM_TriggerTransmit* is called with the relevant *OutGoingPduId* and the content of the PDU is then copied to the buffer given to the function as a reference to the *pdu*. A pduMask is created by using the information from the *IpduMTxPathway*. The PDU is then compared with the expected values, which is calculated with the help of the mask and the default values from the configuration. The test case also handles the Just-In-Time updates by filling out the appropriate stub logs.

IpduMTest_TC_TriggerTransmit test case asserts *IpduM_TriggerTransmit* properly copies the content of its I-PDU transmit buffer into the buffer pointed to by the *PduInfoPtr*, after *IpduM_Transmit* calls have been made. The test is conducted for each *IpduMTxPathway*. The test is only conducted if the *TimerInitialValue* is set to zero. *IpduM_Transmit* calls are made prior to the *IpduM_TriggerTransmit* call, in order to fill up the transmit buffers with the data content of the *inPdu*. The *outPduExpected* is the expected PDU after calling the transmit harness function. This is compared with the *outPduActual*, which is the PDU returned by the *IpduM_TriggerTransmit* function. The test case also handles the Just-In-Time updates by filling out the appropriate stub logs if need be.

The two *TriggerTransmit* test cases fulfill several complex requirements, which are listed below.

- **IPDUM015:** This requirement states that the static part and last received dynamic part are merged in a single I-PDU, with a unique PDU ID, and is then transmitted to the PDU Router. This requirement is fulfilled in the *IpduMTest_TC_TriggerTransmit* test case, because after the transmit functions, the *IpduM_TriggerTransmit* receives the merged I-PDU with the static part and last transmitted dynamic part.

- **IPDUM067:** This requirement states that the internal buffers shall be initialized with the value of *IpduMIPduUnusedAreasDefault* from the configuration. This requirement is fulfilled because when the *IpduM_TriggerTransmit* function is called without any previous *IpduM_Transmit* calls, the data copied from the buffers have the value of *IpduMIPduUnusedAreasDefault*.
- **IPDUM068:** This requirement states the initial values of the dynamic part is set by the stub function called *PduR_IpduMTriggerTransmit*. This requirement is fulfilled since the *PduR_IpduMTriggerTransmit* stub function does exactly this, gives initial value for the dynamic part.
- **IPDUM090:** This requirement states that the *IpduM_TriggerTransmit* function shall copy the contents of the transmit buffer to the buffer given in its argument. This requirement is fulfilled by both test cases when the actual and expected PDU are compared and found to be equivalent.
- **IPDUM143:** This requirement states the initial values of the static part is set by the stub function called *PduR_IpduMTriggerTransmit*. This is fulfilled the exact same way as the IPDUM068 requirement.
- **IPDUM169:** This requirement states that if the *IpduMJitUpdate* is set, then the value of the static and dynamic parts are updated when the *IpduM_TriggerTransmit* function is called.

6.4.3.5 TxConfirmation Test Cases

There are two test cases testing the *IpduM_TxConfirmation* function specifically. Both test cases only run if *ConfirmationEnabled* is set to *TRUE* and the *TimerInitialValue* is set to zero.

IpduMTest_TC_TxConfirmation_Multiple test case examines the outcome of multiple *IpduM_TxConfirmation* calls after an *IpduM_Transmit* call was made. Only the first *IpduM_TxConfirmation* call should be logged, while the rest are ignored. Three individual *IpduM_TxConfirmation* calls are made, in which only the first call generates an actual *TxConfirmation* from the lower layer communication modules, the rest of the *IpduM_TxConfirmation* calls are silently ignored.

IpduMTest_TC_TxConfirmation_Single test case is similar to the previous one, except that an *IpduM_TxConfirmation* call is made after each *IpduM_Transmit* call. This means that each *IpduM_TxConfirmation* call is logged. *IpduMTest_HarnessTransmit_StaticPart* transmits the static part of an I-PDU separately and if the transmission was successful, the *TxConfirmation* expected logs are filled out. *IpduM_TxConfirmation* is called afterwards. The Boolean variable *staticTransmitted* is to indicate that the static part was transmitted. This is taken into consideration in *IpduMTest_HarnessTransmit_DynamicPart.* If the Dynamic Part was transmitted separately then the *TxConfirmation* expected logs are filled accordingly, and afterwards *IpduM_TxConfirmation* is called.

These two test cases fulfill the following requirements:

- **IPDUM022:** This requirement states that if an *IpduM_TxConfirmation* call is made, then the transmit confirmations of the corresponding COM I-PDUs' static and dynamic parts also take place. This requirement is fulfilled when the *TxConfirmation* stub logs are filled out according to the *StaticHandleId* and *DynamicHandleId* in the *IpduMTest_Log_TxConfirmation* function due to the confirmation settings of the static and dynamic part.
- **IPDUM024:** This requirement states that unexpected *TxConfirmations* are ignored silently. This requirement is fulfilled when all but the first *IpduM_TxConfirmation* call is ignored in the test case examining multiple *IpduM_TxConfirmation* calls.
- **IPDUM088:** This requirement states that an *IpduM_TxConfirmation* function call will cause the confirmation of the individual static and dynamic parts of the I-PDU received from the PDU Router. This requirement is fulfilled in a similar manner as IPDUM022.

6.4.3.6 Timeout Test Cases

There are three test cases testing the timeout functionality. The *TxConfirmation* timeout is the time period for which *TxConfirmations* are accepted. Its value is reached through the unsigned 32-bit integer called *TimerInitialValue*. The following test cases are conducted only if the value of *TimerInitialValue* is greater than zero.

IpduMTest_TC_Timeout_Elapsed test case examines that if numerous *IpduM_Transmit* calls are made, only the first one is registered, the rest are ignored, until the timeout elapses. After the timeout has passed, it is possible to successfully call the *IpduM_Transmit* function again. The time passing is simulated by calling the *IpduM_MainFunction*.

IpduMTest_TC_Timeout_TxConfirmation test case inspects the effect of calling the IpduM_TxConfirmation before the timeout elapses. First, an *IpduM_Transmit* call is made and before the timeout can pass, an *IpduM_TxConfirmation* call is made. This confirmation is registered in the stub logs, and a new transmit request is now permitted.

IpduMTest_TC_Timeout_Elapsed_TxConfirmation test case examines if the timeout has passed after an *IpduM_Transmit* call, then the IpduM_TxConfirmation call is ignored. This is because there is no need for confirmation to allow for a new transmission request. The *IpduM_TxConfirmation* function call is not registered in the stub logs.

The following requirements were fulfilled by the timeout test cases.

- **IPDUM020:** This requirement states that if the timeout timer has not elapsed or no confirmation was made, all transmission request are ignored until the timeout passes. This requirement is fulfilled in both the *IpduMTest_TC_Timeout_Elapsed* test case and the *IpduMTest_TC_Timeout_TxConfirmation* test case when the *IpduM_Transmit* function calls are not logged, due to the fact that the timeout has not elapsed or the *IpduM_TxConfirmation* function was not called.
- **IPDUM023:** This requirement states that if no *TxConfirmation* is received before the timeout elapses, then the module shall allow new transmission requests. This requirement is fulfilled in the *IpduMTest_TC_Timeout_Elapsed* test case when new transmissions requests are permitted only after the time has elapsed.
- **IPDUM152:** This requirement states that if the timeout has not elapsed and no confirmation occurred, then any transmission request shall return with *E_NOT_OK*. This requirement is fulfilled in the above mentioned test cases when the return value of the *IpduM_Transmit* function is

compared with the macro *E_NOT_OK*, before the timeout has elapsed. If the two values are the same this requirement is fulfilled.

6.4.3.7 Just-In-Time Update Test Case

A test case was created specifically to test the Just-In-Time update functionality. The Just-In-Time update was mentioned and taken into consideration in previously discussed test cases, but never fully explained in detail. The IpduM module stores the value of the static and dynamic parts, but this information can become outdated. During transmission the parts can be updated with relevant values, by configuring the Just-In-Time update functionality.

IpduMTest_TC_JitUpdate examines the this functionality by calling the *IpduMTest_HarnessTransmit_DynamicPart* harness function and the *IpduMTest_HarnessTransmit_StaticPart* harness function one after the other, while changing the value of the *defaultUsedData*. The *defaultUsedData* is the initial value of the data in the PDU. In this case, it is the value which is used to update the data in the PDU, when the Just-In-Time functionality is invoked. So if the Just-In-Time update is configured in any of the static or dynamic parts, then the value of *defaultUsedData* will be the updated values in the PDU. If this is true for both the expected and actual stub logs, the test case is successful, and the Just-In-Time update functionality is thoroughly tested. This test case is created to fulfill a particular requirement.

- **IPDUM168:** This requirement states that during the transmission of an I-PDU, one of the parts is configured to Just-In-Time update, then the segments of that part will be updated, if the other part triggered the transmission. For example, if the static part of the PDU is transmitted, and the Just-In-Time update of the dynamic part is configured, then the dynamic segments will be updated with new values.

6.5 Generally Tested Requirements

There are few requirements that are not tested by a specific test case or in a particular test suite, but rather by thoroughly testing the module the following requirements can be presumed as fulfilled.

- **IPDUM098:** This requirement was mentioned before, but it is truly only fulfilled if the entire IpduM module is exercise to see it really does not set the value of the selector field.
- **IPDUM101:** This requirement states that the *IpduM_MainFunction* shall act as a scheduling function and perform the processing of the IpduM module. This function is not directly called from the PDU Router like the other API functions of the IpduM module. *IpduM_MainFunction* is the function used to simulate the time passing. In the configuration model, *IpduMConfigurationTimeBase* specifies the time that passes, by one IpduM_MainFunction call. This function is required because the *TxConfirmation* timeout timer needs a way of telling how much time has passed. In the timeout test cases, *IpduM_MainFunction* calls are made to ensure that the timeout elapses. This verifies that it properly simulates time elapsing.
- **IPDUM107:** This requirement states that the IpduM shall not directly access the AUTOSAR OS. The AUTOSAR Operating System cannot be directly accessed by the IpduM module. Testing of this requirement is limited, but since the IpduM module does not make outward function calls toward other modules, and the module behaves properly in a testing environment, it can be said that the AUTOSAR OS is not accessed. In the testing environment only function calls to stubs are made, which guarantees that the AUTOSAR OS is not reached.
- **IPDUM033:** This requirement states that the *IpduM_Init* function shall initialize all the global variables of the IpduM module. During initialization of the IpduM module all the module-related global variables are initialized, meaning that no value of the global variables are kept from before initialization, after calling *IpduM_Init*. The global variables are given new value, in other words they are restored to their original value.

7 Conclusion

After completing the testing environment, and also while producing the test cases, documentation was added. Documentation of the code is crucial to show the thought process that went into the creation of the test cases. This makes it easier to comprehend the source code in case any changes have to be made later. Especially at a company, where numerous people work consecutively on a project, documentation is important to signify changes made in the code. The documentation of the code also includes tagging the requirements in the source code next to the test cases that successfully fulfill these requirements. By running a requirement analyzing tool on the source code, the tags are identified and a benchmark shows the percentage of requirements met. The tool also measures the branch coverage of the IpduM module, which was traversed by running the test cases. After the testable requirements have been fulfilled and the branch coverage of the IpduM module reached 100%, the testing of the module can be deemed finished.

7.1 Contribution

During the course of my thesis, I successfully tested the IpduM module according to the specifications required by the AUTOSAR 4.0 Release. I clearly documented the algorithms of the test cases. The next step in the development of the IpduM module is that it goes into code review. Under code review, my testing environment is inspected for programming formalities and proper documentation. If it is inadequate, according to company standard, it is then handed back to me for correction.

7.2 Aftermath

After the module has undergone code review, possible correction might have to be made to the source code to reach company standards.

Since the IpduM module was tested for requirements given by the 4.0 Release of AUTOSAR, future releases can cause the testing environment to become outdated. If new functionalities are implemented in the module, then the testing environment needs to be adjusted, to satisfy the new requirements. This could mean creating entirely new test cases, or modifying existing ones.

List of Abbreviations

ACC	Autonomous Cruise Control
API	Application Programming Interface
ASIL	Automotive Safety Integrity Level
BC	Branch Coverage
BSW	Basic Software
C	C Programming Language
CAN	Controller Area Network
CC	Condition Coverage
C/DC	Condition Decision Coverage
COM	Communication module
ComM	Communication Manager module
DC	Decision Coverage
DET	Development Error Tracer module
E/E	Electrical and Electronic
ECU	Electronic Control Unit
GCC	GNU Compiler Collection
ID	Identification
IEC	International Electrotechnical Commission
IP	Intellectual Property
I-PDU	Interface Layer Protocol Data Unit
IpduM	I-PDU Multiplexer module
ISO	International Organization of Standardization

LIN	Local Interconnect Network
MC/DC	Modified Condition Decision Coverage
MISRA	Motor Industry Software Reliability Association
OEM	Original Equipment Manufacturer
OSEK	Open System and their Interfaces for the Electronics in Motor Vehicles
OSI	Open Systems Interconnection model
PDU	Protocol Data Unit
PduR	Protocol Data Unit Router module
RTE	Run Time Environment
SW-C	Software Component
TC	Test Case
TKP	ThyssenKrupp Presta AG
TS	Test Suite

Table of Figures

Bibliography

[1] AUTOSAR Consortium, 2012. *Layered Software Architecture.* [Online]
Available at: http://www.autosar.org/fileadmin/files/releases/4-0/software-architecture/general/auxiliary/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf
[Accessed 16 November 2015].

[2] AUTOSAR Consortium, 2015. *Specification of I-PDU Multiplexer.* [Online]
Available at: http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/communication-stack/standard/AUTOSAR_SWS_IPDUMultiplexer.pdf
[Accessed 16 November 2015].

[3] Burden, P., 2013. EE Times Europe Automotive. [Online]
Available at: http://www.automotive-eetimes.com/en/design-center/full-design-center.html?cmp_id=71&news_id=222902845&page=0
[Accessed 20 November 2015].

[4] Johnson, P., 2015. *Testing and Code Coverage.* [Online]
Available at: http://pjcj.net/testing_and_code_coverage/paper.html
[Accessed 23 November 2015].

[5] Jones, N., 2002. *Introduction to MISRA C.* [Online]
Available at: http://www.embedded.com/electronics-blogs/beginner-s-corner/4023981/Introduction-to-MISRA-C
[Accessed 14 November 2015].

[6] Kumar, A. & St. Clair, J., 2005. *CUnit.* [Online]
Available at: http://cunit.sourceforge.net/doc/index.html
[Accessed 15 November 17].

[7] MinGW, 2012. *Minimalist GNU for Windows.* [Online]
Available at: http://www.mingw.org/
[Accessed 28 November 2015].

[8] National Instruments, 2014. *What is the ISO 26262 Functional Safety Standard?.* [Online]
Available at: http://www.ni.com/white-paper/13647/en/
[Accessed 27 November 2015].

[9] OSEK OS/ISO WG, 2005. *OSEK VDX.* [Online]
Available at: http://portal.osek-vdx.org/files/pdf/specs/os223.pdf
[Accessed 10 November 2015].

[10] Pan, J., 1999. *Software Testing.* [Online]
Available at: http://users.ece.cmu.edu/~koopman/des_s99/sw_testing/
[Accessed 22 November 2015].

[11] Patterson, A., 2014. *EE Tims Europe Automotive.* [Online]
Available at: http://www.automotive-eetimes.com/en/autosar-and-iso26262-a-new-approach-to-vehicle-network-design-and-automotive-safety.html?cmp_id=71&news_id=222903893&page=2
[Accessed 14 November 2015].

[12] Vector Informatik GmbH, 2015. *Introduction to AUTOSAR.* [Online]
Available at: https://elearning.vector.com/vl_autosar_introduction_en.druck
[Accessed 19 October 2015].