FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Gerador automático de requisitos e descrições de casos de teste

Roberto Ribeiro Lima

Mestrado em Engenharia Eletrotécnica e Computadores

Orientador: Luís Almeida

Orientador externo: Tiago Rodrigues

22 de Julho de 2018

Gerador automático de requisitos e descrições de casos de teste

Roberto Ribeiro Lima

Mestrado em Engenharia Eletrotécnica e Computadores

22 de Julho de 2018

Agradecimentos

Eu gostaria de agradecer aos meus orientadores, o Professor Luís Almeida e Engenheiro Tiago Rodrigues, pela ajuda, disponibilidade e orientação incansável ao longo deste trabalho, essencial para a realização desta dissertação.

Gostaria também de agradecer à Doutora Sandra Costa pela oportunidade de elaborar a dissertação na BOSCH® *Chassis System Control*, não esquecendo todo o auxílio que me foi prestado durante a realização desta dissertação.

Gostaria de agradecer ao Engenheiro Ivo Brandão, ao Engenheiro David Ribeiro, ao Engenheiro Leandro Ferreira e a todos os colegas da BOSCH® *Chassis System Control*, que direta ou indiretamente me ajudaram no processo de integração da empresa e/ou na elaboração deste projeto.

Resumo

O tema deste documento está relacionado com a industria automóvel, em específico com o desenvolvimento dos sistemas eletrónicos, presentes no automóvel, conhecidos como Electronic Control Units (ECU). ECUs são sistemas embarcados responsáveis por controlar um ou mais sistemas do automóvel. A configuração de um ECU está descrita num arquivo eXtensible Markup Language (XML) (*.arxml) seguindo um esquema definido pela Automotive Open System ARchitecture (AUTOSAR), chamado de ECU *Extract*. Tipicamente, este arquivo é gerado pelo Original Equipment Manufacturer (OEM), ou seja, pelo fabricante do automóvel respetivo.

Neste documento apresentamos uma solução relacionada com Requirements Engineering (RE), referindo-se ao processo de definição, documentação e manutenção de requisitos dos sistemas de *software*. A solução envolve a criação automática de uma lista de requisitos derivados de um ECU *Extract* e a derivação de casos de teste para validar esses requisitos. Um caso de teste é um conjunto de condições de execução e resultados esperados desenvolvidos para um determinado objetivo, como, por exemplo, para executar um determinado caminho do programa ou para verificar a conformidade com uma situação específica.

Com isto, pretende-se que o processo para a criação de requisitos e descrições passo-a-passo dos casos de teste para um ECU *Extract* seja automática, substituindo o processo manual que atualmente é usado. Desta forma é possível tornar os processos de RE, mais eficientes e resilientes a erros humanos.

Salientamos que o objetivo desta aplicação não é realizar testes automaticamente, mas sim fornecer documentos de fácil compreensão. Estes documentos serão usados pelo departamento de testes que, posteriormente, irá realizar os testes e validar vários requisitos extraídos do ECU *Extract*. Serão comparados valores obtidos com os esperados para um conjunto de cenários.

Esta dissertação começará por abordar o desenvolvimento de sistema automotivos de controlo, apresentada os conceitos básicos e as *frameworks* que estão envolvidas. Depois é feita uma análise inicial do projeto diagnosticando o problema, apresentamos uma solução e em seguida são discutidas as metodologias de trabalho utilizadas no desenvolvimento da solução. Por último, apresentamos a arquitetura da solução, os testes realizados para analisar a performance da solução e, finalmente as considerações finais deste trabalho.

Abstract

This document addresses a topic within the automotive industry, focusing on the electronic control units present in a modern automobile, known as ECUs. These ECUs are embedded systems responsible for controlling one or more automobile systems. The ECU configuration is described in an XML file (*.arxml) that follows a schema defined by AUTOSAR called ECU *Extract*. Typically, this file is generated by the OEM, i.e., by the respective automobile manufacturer.

In this document we will present a solution related with Requirements Engineering (RE), thus addressing the definition, documentation and maintenance of the requirements of software systems. The solution provides a list of requirements automatically from the ECU *Extract* together with the description of test cases for such requirements. A test case is a set of execution conditions and expected results to achieve a desired goal, such as checking whether a specific path in the program is executed or checking whether if it is according to a specific requirement.

The requirements and tests case for an ECU *Extract* will be created automatically, replacing the manual process that is currently used. This way, the requirements engineering process can be made more efficient and resilient to human errors.

We emphasize that the focus of our solution is not to execute the tests automatically, but instead, to provide documentation that is easy to undertand by the Test department that after will test and validate the ECU requirements automatically extracted from the ECU Extract, comparing for a set of scenarios, the values obtained with those expected.

This Dissertation will begin by addressing the development of automotive control systems, explaining the basic concepts and frameworks that are involved. Then it will present the problem, the proposed solution and next the working methodologies used in the solution development. Finally, we present the solution architecture, the tests to analyze the performance of the solution and final conclusions.

Conteúdo

Lista de Figuras

Lista de Tabelas

Glossário

ABS Antilock Braking Systems. 6

ADAS Advanced Driver Assistance System. 10

API Application Programming Interface. 14, 15

ARXML AUTOSAR XML. 17, 19, 26, 33, 35, 37, 38, 41, 47, 59

AUTOSAR Automotive Open System ARchitecture. iii, v, 1, 7, 13, 14, 15, 16, 17, 18, 19, 24, 26, 33, 36, 40, 47, 50, 52, 54, 59, 60

BSW Basic Software. ix, 14, 15, 16

BSWM Basic Software Modules. 14

CAN Controller Area Network. vii, ix, 1, 9, 10, 12, 16, 23, 26, 53, 56

CPU Central Processing Unit. 5

CRC Cyclic Redundancy Check. 11

DoIP Diagnostic over Internet Protocol. 12

DOM Document Object Model. ix, 31, 32, 33

ECU Electronic Control Units. iii, v, viii, ix, x, 1, 2, 5, 6, 7, 10, 11, 12, 13, 14, 16, 17, 18, 21, 22, 23, 25, 26, 29, 33, 35, 37, 40, 41, 42, 47, 48, 49, 50, 51, 50, 51, 52, 53, 54, 55, 56, 57, 59, 60

FD Flexible Data-Rate. ix, 10

FTDMA Flexible Time-Division Multiple Access. 8

I/O Input/Output. 5

IEEE Institute of Electrical and Electronics Engineers. 11

IP Internet Protocol. 11, 12

ISO International Organization for Standardization. 9, 12

JAXB Java Architecture for XML Binding. ix, 32, 33, 38, 47, 48, 49, 50, 51, 52

LIN Local Interconnect Network. vii, ix, 12, 23

MAC Medium Access Control. 11

OBD On-board Diagnostics. 10, 12

OEM Original Equipment Manufacturer. iii, v, 7, 10, 13, 16, 22

OSI Open System Interconnection. 9, 11

PTP Precision Time Protocol. 43, 55, 57

RAM Random-Access Memory. x, 50, 52

RE Requirements Engineering. iii, 1, 2, 21, 23, 25, 36, 55

RTE Runtime Environment. 14, 15, 16

SAX Simple API for XML. 31, 32, 33

SCI Serial Communication Interface. 12

SW-C Software Component. ix, 15, 16, 15, 16

TDMA Time-Division Multiple Access. ix, 8

UML Unified Modeling Language. 31, 35, 45

UTF Unicode Transfomation Format. 17

VLAN Virtual Local Area Network. 11

V-model Validation & Verification model. ix, 19, 20

WiP Work in Progress. 28

WWH-OBD World Wide Harmonize-On-board Diagnostics. 12

XML eXtensible Markup Language. iii, v, vii, 1, 2, 17, 18, 19, 30, 31, 32, 33, 35, 38, 40, 43, 48, 49, 51, 52, 59

Capítulo 1

Introdução

Ao longo dos últimos anos tem-se verificado uma evolução considerável na industria automóvel. Este meio de transporte continua a ser o mais utilizado no quotidiano das pessoas e os fabricantes de automóveis, juntamente com fabricantes de componentes eletrônicos, tentam tornar as viagens o mais confortáveis possível. Cada vez mais é possível encontrar em automóveis sistemas de controlo capazes de fornecer e integrar diversos serviços para o bem estar dos passageiros, como por exemplo, monitorizar a condução ou o interior do carro.

Com a evolução da tecnologia, a complexidade dos sistemas de controlo tem vindo a aumentar devido ao facto de cada vez mais os sistemas serem interdependentes. Estes sistemas estão suportados em unidades de controlo cada vez mais poderosas e complexas, conhecidas como ECUs. Devido à elevada complexidade que existe em projetar estes sistemas, várias empresas ligadas à indutaria automóvel e à industria eletrónica juntaram esforços. Como resultado destas parcerias surgiram padrões para o processo de definição, documentação e manutenção dos requisitos no desenvolvimento das aplicações e dos ECUs. Assim, surgiu a *framework* AUTOSAR que fornece aos projetistas uma visão mais abstrata do *hardaware* em relação à aplicação e *guidelines* para o processo de desenvolvimento. Os padrões para o processo de documentação e manutenção dos requisitos enquadram-se no domínio da RE, que consiste em definir regras na descrição do sistema, como a criação dos requisitos que o sistema tem de cumprir.

Segundo a *framework* AUTOSAR, a descrição do comportamento do sistema de controlo (ECU) é feita num documento XML conhecido como ECU *Extract*. Neste documento estão descritas as especificações técnicas do sistema de controlo. Sempre que este sofre *updates* ao serem acrescentadas novas funcionalidades, o documento respetivo tem de ser atualizado. Com base neste documento são estabelecidos os requisitos do sistema, ou seja, são descritas em forma de requisitos as funcionalidades e restrições que o sistema tem de cumprir. Por exemplo, vejamos um sistema que monitoriza a posição do automóvel utilizando o protocolo CAN para comunicar com outros sistemas. Assim, um requisito e também restrição do sistema é ter a capacidade de estabelecer comunicação com outros sistemas através de uma porta de comunicação CAN. Este requisito está especificado no ECU *Extract* do sistema, mas aparece inserido num documento que é longo e complexo, que facilmente pode passar despercebido no momento de gerar os requisitos de teste

do ECU. Por causa desta dificuldade há a necessidade de elaborar documentação que traduza o conteúdo do ECU *Extract* para um formato objetivo e de fácil compreensão, entrando assim no dominio de RE. Sempre que este sistema de monitorização sofre alterações é necessário refazer a documentação, relativa aos requisitos dos sistema e dos casos de teste para validar os requisitos. Para além do sistema possuir requisitos, uma parte importante do desenvolvimento é a validação das funcionalidades e se são cumpridas as restrições descritas nos requisitos. Este processo de validação consistes em criar cenários onde são testadas os requisitos de forma exaustiva. Neste exemplo, deverão ser criados cenários para testar a comunicação CAN.

Os processos de criação de requisitos e casos de teste são feitos manualmente, ou seja, é necessário olhar diretamente para o código XML e traduzir a informação para uma linguagem clara e sucinta as restrições, bem como as descrições dos cenários para validar o sistema. Esta tarefa é demorada e por se tratar de um processo manual obriga a gasto de recursos humanos. Num ambiente industrial, onde o fator tempo é crucial, este método de trabalho não é muito produtivo. Por isso, existe a necessidade de tornar estes processos automáticos libertando recursos humanos e tentar otimizar a gestão de tempo.

Esta dissertação decorreu no âmbito de uma parceria com a BOSCH® *Chassis System Control*, em Braga, com o objetivo de estudar uma alternativa ao processo manual de criação de requisitos e descrição de casos de teste. Para isso é proposto uma solução que permita de uma foram automática extrair do ECU *Extract* os requisitos do sistema e gerar descrições de casos de teste para validar os mesmos.

1.1 Objetivos

O principal objetivo do trabalho descrito nesta dissertação é criar uma solução capaz de obter informações técnicas do ECU *Extract*, nomeadamente a configuração de um ECU *Extract* e, elaborar um conjunto de requisitos e de descrições passo-a-passo de casos de teste que permitem validar os requisitos. Assim sendo, a solução deverá ser capaz de filtrar as informações do arquivo, selecionar e relacionar os dados relevantes e produzir requisitos funcionais e não funcionais relacionados com a performance do sistema. Uma vez que a solução a desenvolver irá enquadrar-se num ambiente industrial é fundamental que a mesma possa vir a acompanhar as necessidades dessa indústria. Desta forma, a solução proposta deve ser escalável, sendo capaz de incorporar novas funcionalidades sem a necessidade de fazer uma reestruturação da sua arquitetura. Por fim, esta deve também ser capaz de integrar uma cadeia de ferramentas que fornece suporte adicional à validação do sistema.

1.2 Resumo dos Resultados

Na sequência deste trabalho conseguimos produzir uma aplicação que lê e valida os documentos do ECU *Extract* (.arxml) segundo um esquemático (.xsd). Após esta validação é filtrada a informação de maior relevância na descrição no ECU *Extract* para elaborar automaticamente os

requisitos do sistema de controlo. Os requisito gerados pela aplicação seguem determinados padrões de escrita, evitando que haja ambiguidade na definição do requisito. No final a aplicação vai apresentar ao utilizador um documento *Excel* com a lista de requisito dos ECU e outro documento *Excel* com a descrição de casos de teste para validar os requisitos gerados. Assim, esta aplicação permite automatizar os processos de RE tornando-os também mais eficientes e resilientes a erros humanos.

1.3 Estrutura do documento

Esta dissertação está estruturada da seguinte forma. O Capitulo seguinte oferece uma visão geral sobre a indústria automóvel, nomeadamente os sistemas de controlo, os protocolos de comunicação utilizados, ferramentas utilizadas no desenvolvimento e testes de validação dos sistemas.

No Capítulo 3 são discriminadas as etapas iniciais e preparações para elaborar a solução proposta neste documento. Estas processos incluem a metodologia de trabalho, o planeamento e o levantamento das ferramentas a usar.

No Capítulo 4 descrevemos a estrutura da solução proposta, nomeadamente o diagrama de classes, o processo para gerar requisitos e casos de teste.

No Capítulo 5 apresentamos a descrição dos resultados obtidos bem como a discussão das opções escolhidas e uma análise temporal da solução.

Por fim, no capítulo 6 são ditadas as conclusões do projeto e enumeradas as possíveis implementações futuras.

Capítulo 2

Sistemas de controlo em automóveis

Atualmente os automóveis são máquinas computacionais com um vasto número de funções para além da função básica de transporte. Nos automóveis estão presentes vários sistemas como o motor, a transmissão, sistema de suspensão ativo, sistema de travagem, etc.. Para além destes sistemas diretamente relacionados com a função de transporte existem outros, cujo o foco é a segurança e o conforto dos utilizadores (Kiencke e Nielsen, 2005). Um ou mais micro-controladores são utilizados na gestão destes sistemas formando assim um sistema de controlo que permitem lidar com características não lineares, parâmetros variantes no tempo e variáveis dinâmicas. Contudo, quando se pretende tratar inúmeras variáveis, em que algumas são difíceis ou até impossíveis de monitorizar, os sistemas de controlo tornam-se um grande desafio de engenharia (Kiencke, 1988).

Na eletrónica automóvel, um ECU é um sistema embarcado cuja função é controlar um ou mais sistemas existentes no automóvel. Um ECU contém um Central Processing Unit (CPU), a memória, Input/Output (I/O) específicos e as comunicações necessárias para a integração no sistema global.

2.1 Sistemas centralizados e distribuídos

Os sistemas de controlo nos automóveis podem seguir uma abordagem centralizada ou distribuída dependendo da forma como processam e executam as operações de controlo.

Num sistema centralizado existe um ECU central que recebe os dados, processa-os e gera os *outputs*. Uma vez que o ECU é responsável por todo o funcionamento do sistema isto implica, também, que possua todo o *software* e *hardware* para ler os *inputs* e gerar os sinais de *output*. Esta propriedade dos sistemas centralizados torna-se uma grande desvantagem porque limita a escalabilidade do sistema, uma vez que o número de I/Os que o ECU central pode tratar é limitado. No entanto, a principal vantagem deste tipo de sistema reside na simplicidade da sua arquitetura sendo o *hardware* apenas constituído por um ECU que contém os I/O necessários para ler sensores e controlar atuadores.

Por outro lado, nos sistemas distribuídos, a função do sistema pode ser distribuída por vários ECUs, ou seja, diferentes ECUs têm sub-funções e trabalham em conjunto para realizar uma função principal. Desta forma, é possível obter uma redução na capacidade de processamento que é necessária no ECU uma vez que a função principal está repartida em pequenas funções e é executada por múltiplos ECUs. No entanto, em sistemas distribuídos a complexidade da arquitetura é mais elevada devido ao aumento do número de ECUs e à necessidade da existência de módulos de comunicação. Esta complexidade pode levar a atrasos nas operações de controlo devido ao facto de ser necessária a comunicação entre ECUs[1].

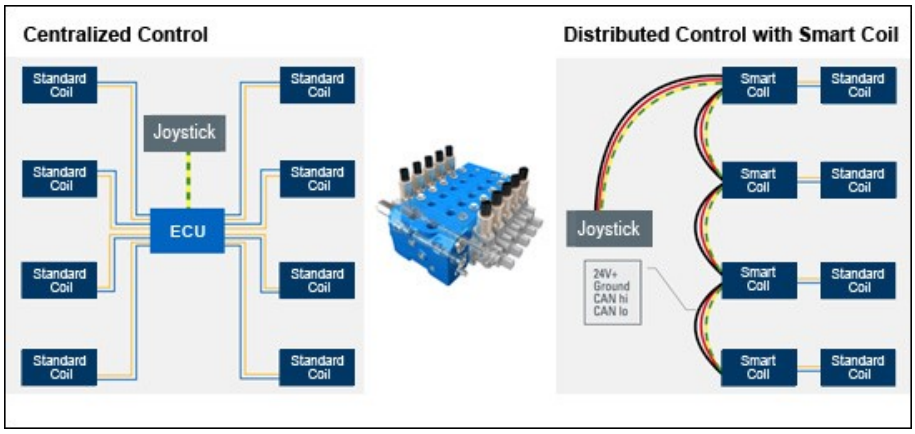


Figura 2.1: Sistema de controlo centralizado *versus* distribuído [2].

Na figura 2.1 está representado o sistema de controlo de um automóvel que pode seguir uma estrutura centralizada ou distribuída. Na versão centralizada, o único poder computacional está no ECU central (Figura 2.1 lado esquerdo) que está responsável por todos as operações de controlo (*I/O*). No sistema distribuído (Figura 2.1 lado direito) já não existe o ECU central sendo as operações de controlo distribuídas por vários ECU (representados pelas caixas *smart coil*). Assim, o controlo do sistema pode ser feito por vários ECUs com menor capacidade do que o ECU central do sistema centralizado.

Em suma, a nível prático, uma arquitetura centralizada é indicada para sistemas de controlo mais simples. Por outro lado, quando se trata de sistemas maiores, a arquitetura distribuída pode ser facilmente estendida com mais ECUs (com menos poder de processamento e *I/O*) resultando num custo total de implementação menor.

Há algumas décadas atrás, o número de sistemas eletrónicos que se encontravam num automóvel era relativamente baixo e de baixa complexidade. Como tal, não havia necessidade de seguir uma arquitetura distribuída. No entanto, o número de ECUs tem vindo a aumentar devido à implementação de sistemas avançados cujo objetivo é aumentar o conforto e a segurança do utilizador, e.g. sistema de Antilock Braking Systems (ABS), *voice control* e monitorização do

[1] By Chris Schottler, Engineering Manager—Advanced Technology Team, Eaton Hydraulic Group URL http://www.eaton.com/Eaton/dmc/ArticleArchive/distributed-vs-centralized-control/index.htm

[2] "Eaton Hydraulic Group"URL http://www.eaton.com/Eaton/dmc/ArticleArchive/distributed-vs-centralized-control/index.htm"

comportamento do condutor (Navet et al., 2007). A elevada complexidade destes sistemas exige um elevado número de ECUs o que origina problemas relacionados com a capacidade de gestão e custo da produção.

Em alternativa, com o surgimento de ECUs mais potentes, baseados em processadores *multicore* (processadores com múltiplos núcleos de processamento), os OEMs conseguiram diminuir o número de ECUs necessárias, visto que estes processadores são capazes de executar mais funções. Nesta situação, cada núcleo do processador *multicore* integra a funcionalidade que um ECU mono processador desempenharia em domínios semelhantes ou iguais. O desafio agora é adaptar os métodos do projeto aos processadores *multicore*, principalmente ao nível de *software*, uma vez que a introdução deste tipo de arquitetura vai implicar alterações drásticas no *software* do ECU (Senthilkumar e Ramadoss, 2011).

Uma prova do sucesso de ECUs baseados em processadores *multicore* é o facto de o próprio consórcio AUTOSAR, em 2011, ter tomado a iniciativa de estabelecer vários padrões que incluem "*guidelines*" para projetos centralizados e distribuídos com um pequeno número de ECUs baseados em processadores *multicore*. Neste caso, os recursos de um ECU, ou vários ECUs conectados em rede, podem ser partilhados na execução de diferentes funções (Senthilkumar e Ramadoss, 2011).

2.2 Protocolos de comunicação na indústria automóvel

Desde a década de 1990 que a indústria automóvel começou a usar arquiteturas distribuídas dando grande atenção aos protocolos de comunicação. Estes protocolos desempenham um papel fundamental na comunicação entre os ECUs pois são utilizados na coordenação entre as unidades de controlo e os recursos físicos. A escolha do protocolo de comunicação vai estar diretamente relacionada com os requisitos de comunicação do sistema, tais como: largura de banda, gestão de transmissões, tolerância a falhas, flexibilidade e segurança.

De seguida são apresentados alguns dos protocolos de comunicação entre ECUs mais utilizados pela industria automóvel.

2.2.1 *FlexRay*

O protocolo *FlexRay* foi proposto por vários fabricantes e fornecedores como um protocolo híbrido. Este protocolo permite partilhar o barramento para mensagens orientadas a eventos e mensagens síncronas, oferecendo as vantagens de cada tipo de mensagem.

Um sistema de comunicação que implementa o protocolo *FlexRay* vai ser composto por vários nós e um meio físico para estabelecer as ligações entre todos os nós (*FlexRay Bus*). A comunicação não está restrita a uma topologia específica, a transmissão pode ser baseada em diferentes topologias, tais como, comunicação ponto a ponto, em linha ou em estrela, sem influenciar a viabilidade da comunicação.

Relativamente à tolerância a falhas, o *FlexRay* permite fazer a comunicação por um canal (*Single Channel*) ou por dois canais (*Dual Channel*). A diferença entre os dois modos de comunicação

é que no caso da comunicação ser por *Dual Channel* é permitido transmitir mensagens críticas com replicação nos dois canais. O *Dual Channel* também permite aumentar a troca de dados para 20 Mbit/s, em vez dos 10 Mbit/s, no caso de se usarem os dois canais de forma complementar.

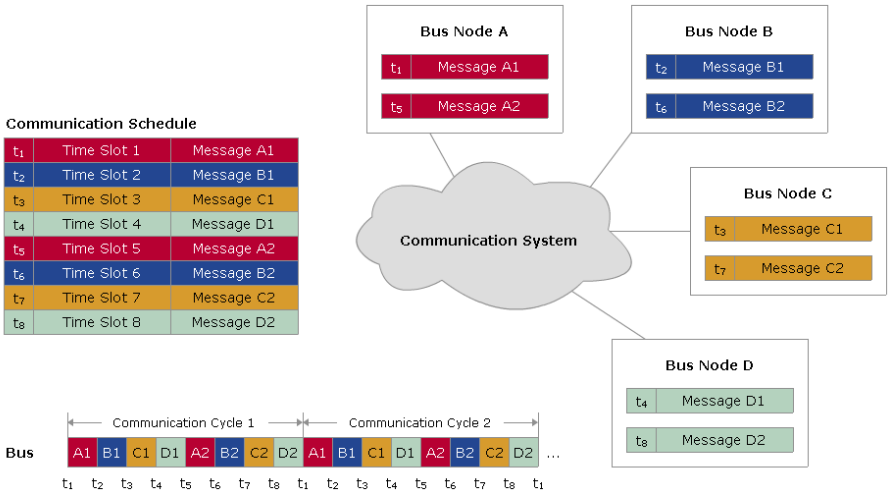


Figura 2.2: Troca de mensagens usando TDMA[3].

Na Figura 2.2 está ilustrado o fluxo das mensagens de um sistema baseado em *FlexRay* utilizando comunicação síncrona. O sistema é composto por quatro nós, onde cada nó tem duas mensagens agendadas. A ordem das mensagens no barramento vai depender do *slot* temporal definido na mensagem de cada nó.

Neste protocolo existem dois modos de garantir o acesso do nó ao barramento, sendo eles o método TDMA ou Flexible Time-Division Multiple Access (FTDMA). Estes métodos garantem que os nós não acedem ao barramento de forma desordenada, implementado uma conformidade entre os nós e garantindo que estão sincronizados com o intervalo de tempo específico para cada mensagem.

O método TDMA segue um cronograma para a comunicação que divide a linha temporal em vários *slots* de tempo de igual duração (*slots* estáticos). Os *slots* temporais são distribuídos pelo numero total de nós, cada *slot* é atribuído a um nó. Durante a comunicação, o nó recebe acesso ao meio de comunicação (barramento) de acordo com o cronograma. Do primeiro ao último *slot* estático o acesso ao barramento é exclusivo, não sendo possível o acesso simultâneo ao barramento por dois ou mais nós.

O cronograma de comunicação é executado periodicamente obrigando a que todas as mensagens estáticas sejam transmitidas com um período especifico. Isto torna o processo de comunicação determinístico sendo possível determinar a mensagem que está a ser enviada e que nó tem acesso ao barramento.

O método TDMA não é a solução ideal para mensagens assíncronas ou esporádicas. Porém, para estes casos, existe a possibilidade de os nós acederem ao barramento de forma mais dinâmica, através de uma comunicação baseada em eventos usando o FTDMA. A diferença entre o método

[3] "Vector:E-LEARNING- Introduction to FlexRay"URL https://elearning.vector.com/index.php?wbt_ls_kapitel_id=490463&root=378422&seit

FTDMA e o TDMA é o facto de o primeiro permitir criar *slots* dinâmicos, permitindo transmitir mensagens apenas quando há necessidade. Contudo, isto significa que o momento da transmissão da mensagem não é previsível e podem existir nós que pretendem enviar mensagens mas não podem transmitir no ciclo atual (Nolte et al., 2005; Mayer, 2006a).

2.2.2 CAN

O protocolo CAN é a rede de transporte mais usada na industria automóvel. Ao longo dos anos diversas aplicações de CAN foram desenvolvidas e usadas em diferentes cenários. Esta popularidade do protocolo deve-se principalmente à facilidade de análise e implementação.

O protocolo é um *standard* internacional desde 1994 e é descrito por quatro documentos da International Organization for Standardization (ISO)(ISO 11898-1, ISO 11898-2, ISO 11898-3 e ISO 11898-4) sendo referente à camada *Data Link* e à camada física do modelo Open System Interconnection (OSI). Contudo, este protocolo pode ser complementado de diferentes formas, dependendo do modo como as mensagens são tratadas na camada *Data Link* (Nolte et al., 2005). Na prática, a rede é baseada numa topologia em barramento onde as unidades de controlo estão ligadas (Figura 2.3) através de dois fios. Nas extremidades do barramento da rede são utilizados resistências (*termination resistors*) para evitar fenómenos transitórios (reflexões). A transmissão do sinal na camada física de uma rede CAN é baseada na transmissão de tensões diferenciais. O meio de transmissão consiste em duas linhas: CAN *high line* (CANH) e CAN *low line* (CANL). A transmissão em modo diferencial permite reduzir a suscetibilidade a interferência eletromagnética. Este protocolo tem uma taxa de transmissão máxima de 1 Mbit/s, com uma extensão máxima na cablagem de cerca de 40 metros e com número máximo de 32 nós (Standard, 1993). Maiores distâncias são possíveis mas com uma correspondente redução da taxa de transmissão. Por exemplo, a 50 κbit/s é possível estender o barramento até 1km.

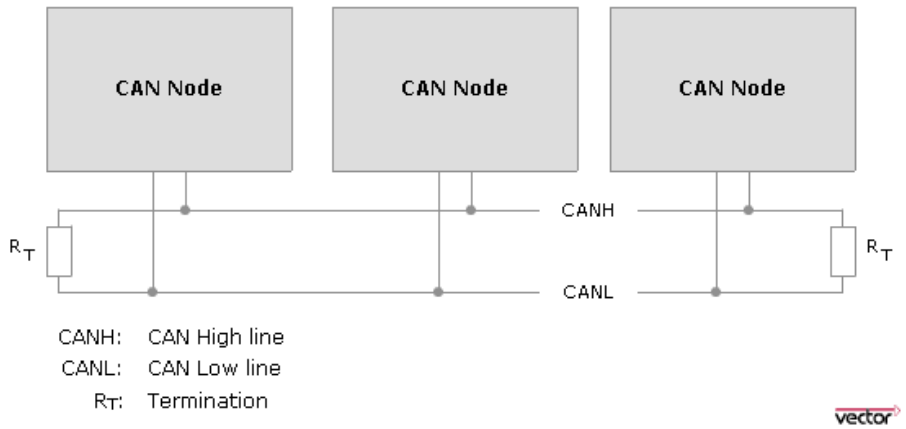


Figura 2.3: Barramento CAN seguindo a topologia em linha[4].

Em aplicações onde o requisito de segurança é a prioridade não é recomendado utilizar apenas um nó (*master*) para controlar o acesso ao barramento porque a falha deste nó causaria a falha

[4] "Vector:E-LEARNING- Introduction to CAN"URL https://elearning.vector.com/index.php?&wbt_ls_seite_id=489572&root=378422&

em toda a comunicação. Por isso, é aconselhado usar um barramento descentralizado, ou seja, qualquer nó tem permissão para aceder ao barramento. Esta é precisamente uma das vantagens do protocolo CAN onde uma arquitetura *multi-master* é implementada numa topologia em barramento que permite que qualquer nó possa enviar mensagens independentes. Neste protocolo, não existe qualquer regra que pré-defina estaticamente a sequência de mensagens ou tempo de transmissão porque a transmissões são orientadas a eventos. Quando surge um evento e o canal está desocupado dá-se inicio à transmissão da mensagem, colocando o canal como ocupado.

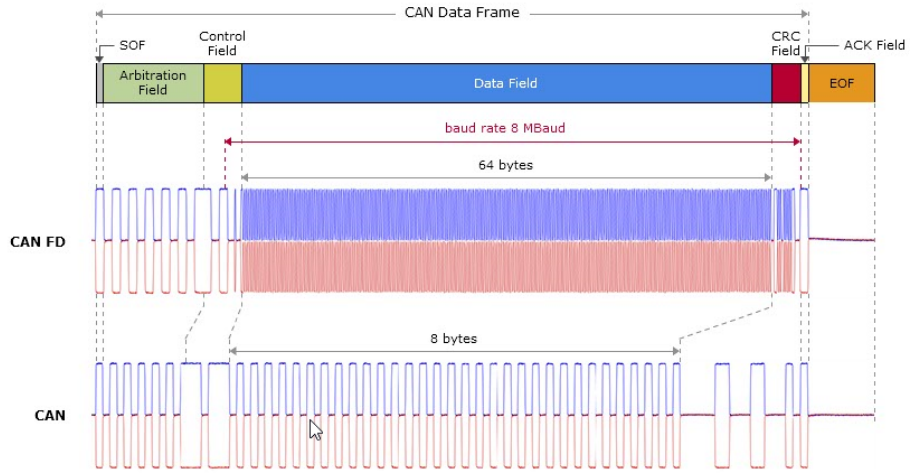


Figura 2.4: Diferença entre a *frame* CAN e CAN FD[5].

Contudo, uma das desvantagens do CAN é a baixa taxa de transmissão do protocolo (1Mb/s no máximo) que limita a quantidade máxima de informação que se consegue transmitir por unidade de tempo (Christmann, 1993). Para combater este problema foi criada recentemente uma extensão do CAN, o CAN FD que permite taxas de transmissão superiores. Esta extensão, permite também controlar a taxa de transmissão durante a própria transmissão da mensagem, sinalizada através do campo *Control Field* da *Data Frame* (Figure 2.4). Assim sendo, CAN FD permite uma transmissão mais rápida na parte da *frame* correspondendo à transmissão dos dados, reduzindo o tempo de transmissão e, consequentemente, permitindo mais carga no barramento.

Para aplicar o CAN FD são necessários novos controladores substituindo os controladores tradicionais CAN. No entanto, estes novos controladores são compatíveis com o CAN tradicional permitindo ser usados tanto para CAN FD como CAN (Decker, 2013).

2.2.3 *Ethernet*

Um protocolo utilizado por várias OEMs da indústria automóvel (e.g. BMW® e Daimler AG®) e fabricantes de componentes eletrónicos (e.g. BOSCH® e Continental®) é *Ethernet* (Kern et al., 2011). O interesse em se utilizar este protocolo proveniente das comunicações de dados genéricas deve-se ao aumento constante dos requisitos de largura de banda nos sistemas automotivos associados à introdução de sistemas de apoio à condução, conhecidos como Advanced Driver Assistance

[5] "Vector:E-LEARNING- Introduction to CAN"URL https://elearning.vector.com/index.php?seite=vl_can_introduction_en&root=378422&wbt

System (ADAS) e On-board Diagnostics (OBD). Hoje em dia, com o aumento das funcionalidades avançadas de *software* presentes no automóvel, o OBD torna-se um sistema necessário para várias funções como monitorização das emissões de gás, diagnóstico dos componentes e das suas propriedades, serviços e manutenção dos sistemas e *download* e *updating* de *software*. O *updating* do sistema quando este já se encontra montado permite uma redução de custos e de tempo na reprogramação dos ECU durante a produção ou na oficina (Bello, 2011).

Ethernet, IEEE 802.3, é um dos *standards* mais utilizados para redes locais baseada no envio de pacotes. Este *standard* tem sofrido constates atualizações, com o constante aumento da velocidade da comunicação de dados. Os *standards* Ethernet define o encapsulamento de sinais, o formato de pacotes e protocolos para a sub-camada de controle de acesso ao meio (Medium Access Control (MAC)) do modelo OSI (Santitoro, 2003). *Ethernet* não é apenas um protocolo mas uma coleção de diferentes *standards* gerados pelo Institute of Electrical and Electronics Engineers (IEEE) que é a entidade responsável pela manutenção e desenvolvimento destes *standards* desde 1980, fornecendo a descrição do comportamento na camada física e a camada *Data Link* do modelo OSI.

Para estabelecer as ligações entre os nós são usados cabos com pares entrançados e as transmissões usam tensões diferenciais simétricas (© 2010-2018 Vector Informatik GmbH, 2014).

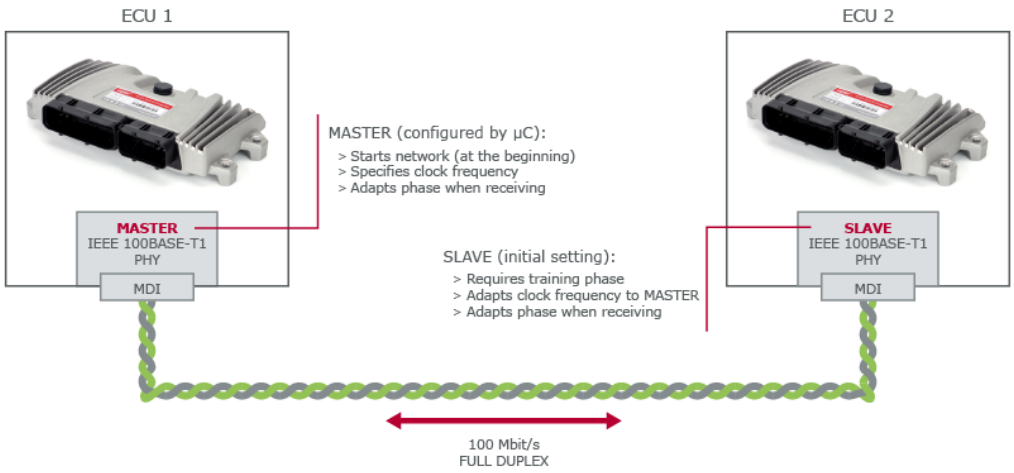


Figura 2.5: Conexão de dois ECU através de *Ethernet*[6].

Todavia, a topologia utilizada é ponto-a-ponto para a conexão. Para ligar mais do que dois nós é necessário utilizar um elemento de acoplamento, normalmente um *switch*.

Alguns *standars* permite que a troca de mensagens entre ECU seja *full-duplex*, ou seja, dois ECUs podem enviar e receber mensagens em simultâneo sem qualquer interferência entre si (Figura 2.5). Existem vários formatos para a *frame* da *Ethernet* definidos pelo IEEE. No entanto, a *frame* geralmente usada na industria automóvel é a *frame Ethernet* II, podendo ter informações de Virtual Local Area Network (VLAN) como uma extensão (Figura 2.6).

[6]"Vector:E-LEARNING- Automotive Ethernet"URL https://elearning.vector.com/index.php?wbt_ls_kapitel_id=1588372&root=378422
[7]"Vector:E-LEARNING- Automotive Ethernet"URL https://elearning.vector.com/index.php?&wbt_ls_seite_id=1588396&root=378422

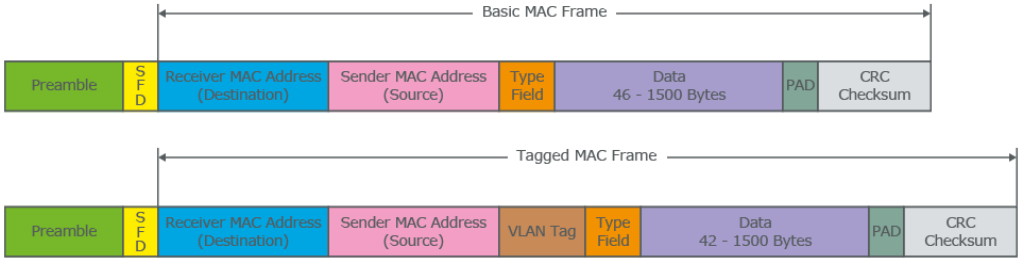


Figura 2.6: *Frame Ethernet* II[7].

A MAC Frame tem início com o endereço do destino identificando o nó da rede que deve receber a mensagem e, só depois, o endereço do remetente. A mensagem enviada pelo remetente pode ser do tipo *Unicast*, *Multicast* ou *Broadcast* permitindo que a mesma mensagem chegue a um ou a vários nós destino. Após os campos com os endereços de destino e de origem há um campo que define o tipo de protocolo (*Type filed*). Este campo normalmente identifica o pacote contido no campo de dados (*Data*) e fornece informação dos protocolos usados na camada superior, e.g. IPv4 ou IPv6. No entanto, se o valor do campo *Type filed* for 0x8100, este campo é deslocado quatro *bytes* para direita porque isto significa que os quatros *bytes* seguintes correspondem à *Tag* VLAN. No fim da *frame* existe o campo Cyclic Redundancy Check (CRC) utilizado no recetor para verificar a integridade da mensagem. Caso haja necessidade de trocar mensagens entre nós de diferentes redes é necessário utilizar o protocolo Internet Protocol (IP) que permite a comunicação para além dos limites da rede local.

As Nações Unidas iniciaram uma ação para estabelecer um padrão para o OBD seguindo o modelo ISO, criando assim o World Wide Harmonize-On-board Diagnostics (WWH-OBD) cujo principal objetivo reside na substituição de padrões regionais de diagnóstico de veículos no controlo das emissões de gás (Bello, 2011; Bruckmeier, 2010). Este *standard* também é conhecido como Diagnostic over Internet Protocol (DoIP) e utiliza *Ethernet* na sua camada física. O DoIP *standard* segue as especificações ISO13400 e promove o uso do protocolo IP para diagnóstico e o uso do protocolo *Ethernet* como substituto do CAN na reprogramação e diagnóstico dos ECUs (Lang, 2010).

2.2.4 LIN

Uma rede LIN consiste num conjunto de nós que estão ligados entre si através de um barramento controlado por um *master* que determina quando os restantes nós, os *slaves*, devem comunicar. Por razões de custo, as funções *master* e *slave* são implementadas em *software*, por cima de um interface de comunicação série (Serial Communication Interface (SCI)).

A transmissão do sinal usa apenas uma fio, estando limitada a uma taxa de transmissão máxima de 20 κbit/s, por suscetibilidade a ruído eletromagnético. Outra restrição neste protocolo é o número de nós na rede onde é recomendada a utilização de no máximo 16 nós. A transmissão

[8] "Vector:E-LEARNING- Introduction to LIN Ethernet"URL https://elearning.vector.com/index.php?seite=vl_lin_introduction_en&root=37842

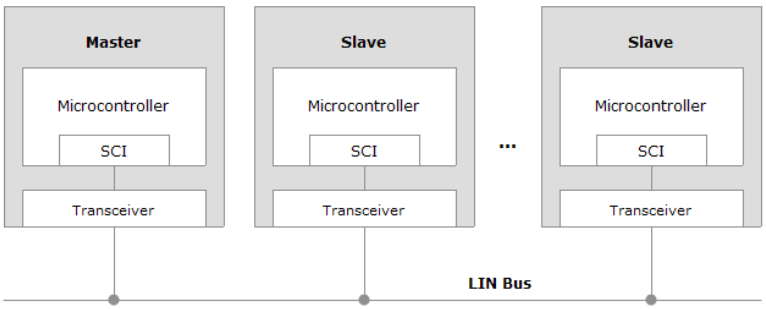


Figura 2.7: Rede LIN[8].

ocorre através da SCI e é orientada por *bytes*, começando a transmissão a partir do *byte* menos significativo.

Como a comunicação é baseada na arquitetura *master-slave*, quem controla e dá início à comunicação é o *master*. Além disso, este também delega a ordem que *slaves* acedem ao barramento através do *Delegated Token*, tornando assim a comunicação determinista.

O facto do barramento ser controlado por apenas um *master* cria uma desvantagem. No caso do *master* falhar toda a comunicação falha, o que não é adequado para comunicações críticas a nível de segurança. Por se tratar de uma comunicação que não é orientada por eventos, ou seja, o *master* tem sempre de enviar um pedido para o *slave* enviar a mensagem, não é possível o *slave* enviar mensagens de forma autónoma para o barramentos. Para contrariar esta desvantagem o protocolo foi estendido para permitir mensagens adicionais (Mayer, 2006b).

2.3 AUTOSAR

Numa tentativa de estruturar o desenvolvimento de *software* para os sistemas automóveis, em 2003, as entidades relacionadas com este sector formaram uma parceria mundial de desenvolvimento para criar uma *framework* conhecida por AUTOSAR. Esta parceria teve como objetivo criar e estabelecer uma arquitetura de *software open-source* e normalizada para ECUs de forma que seja possível maximizar a reutilização de código para diferentes variantes do automóvel e plataformas, transferência de *software*, aumentar a disponibilidade e segurança, fomentar a colaboração entre várias entidades desde fornecedores de equipamento, de serviços e os OEMs (Heinecke et al., 2004).

2.3.1 Arquitetura

A *framework* AUTOSAR fornece uma visão abstrata do sistema, não fazendo suposições sobre a distribuição ou mapeamento de *software* em recursos, numa fase de desenvolvimento inicial. Esta abordagem permite ao projetista uma grande facilidade de uso e liberdade em relação às configurações dos sistemas envolventes. Por outro lado, esta abordagem também implica um alto nível de abstração do modelo de aplicação face à implementação (Piper et al., 2012). Estes níveis de abstração estão divididos em três camadas:

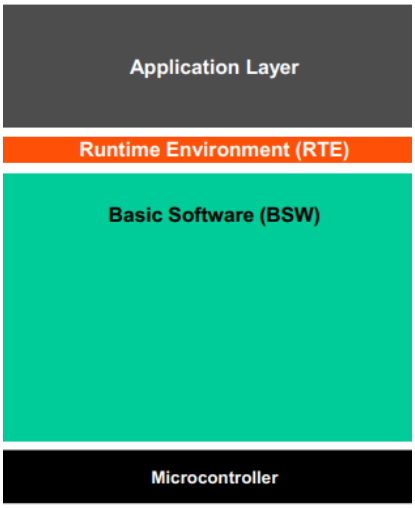


Figura 2.8: Arquitetura AUTOSAR (Piper et al., 2012).

- ***Application Software*:** Trata-se do nível mais alto de abstração, focando-se apenas na funcionalidade principal do sistema para o qual é projetado;
- ***Runtime Environment (RTE)*:** É responsável pela comunicação e interação com a camada superior, fornecendo uma Application Programming Interface (API) adequada;
- ***BSW*:** Consiste em Basic Software Modules (BSWM), ou seja, são um conjunto de arquivos de *software* (código e descrição) que define uma determinada funcionalidade do *software* presente num ECU. Os vários BSWM do BSW podem ser desenvolvidos de forma independente.

A camada de aplicação (*Application Software*) é a camada mais independente em relação ao *hardware*, esta camada abstrai-se completamente do *hardware* e está projetada como uma composição de componentes. À medida que se vai descendo nas camadas essa independência vai diminuindo. A comunicação entre os componentes ou o acesso à camada de BSW é feita através da camada RTE. Esta camada é responsável por toda a interação com a camada de aplicação. Na camada mais baixa temos a camada de BSW , sendo a camada que está em contacto direto com o micro-controlador.

Entrando em mais detalhe na arquitetura do AUTOSAR, a camada do BSW é dividida em vários grupos funcionais e cada grupo está responsável por uma ou várias funções do sistema. Essas funções podem ser, ainda de uma forma abstrata, agrupadas em 4 grupos (Figura 2.9):

- ***Microcontroller Abstraction Layer*:** Representa o nível mais baixo de *software* dentro da camada BSW. Esta camada contém os *drivers* internos (módulos de *software*) com acesso direto ao micro-controlador. A tarefa desta camada é tornar a camada superior independente em relação ao micro-controlador.
- ***ECU Abstraction layer*:** Esta camada interage com os *drivers* da camada anterior (*Microcontroller Abstraction Layer*) e contém os *drivers* para a comunicação com dispositivos

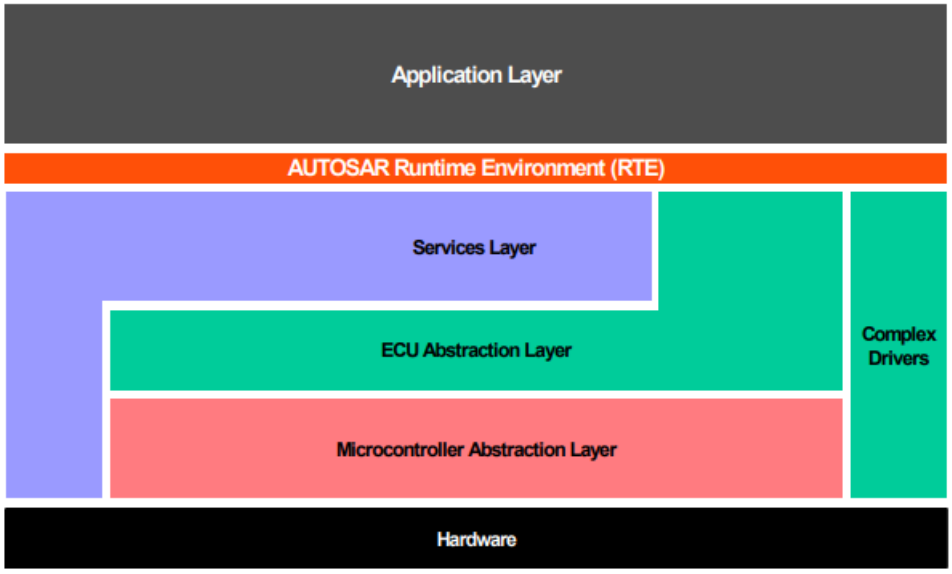


Figura 2.9: Arquitetura AUTOSAR- Divisão da camada BSW(Piper et al., 2012).

externos. Fornece também uma API para o acesso a periféricos e/ou a dispositivos independentemente da sua localização (micro-controlador interno/externo) e as suas respetivas conexões (e.g. tipo de interface) tendo a responsabilidade de tornar as camadas superiores de *software* independentes do *layout* do *hardware* do ECU.

- ***Services layer*:** Trata-se da camada mais alta dentro do BSW, tendo grande relevância na relação com o *software* da aplicação. Esta camada oferece os serviços básicos para a aplicação, para a camada RTE e para os modelos de *software*.

- ***Complex Drivers*:** É a camada responsável pelo elo de ligação entre o *hardware* e a camada RTE. Para além dessa função, permite a possibilidade de integrar funcionalidades especiais tais como, *drivers* para dispositivos que não são especificados pelo AUTOSAR ou dispositivos com restrições temporais muito apertados.

Ainda dentro de cada sub-camada da camada BSW é possível fazer a separação em módulos responsáveis por um conjunto de funções que podem ou não ser utilizadas (Figura 2.10). Isto garante a capacidade modular e a arquitetura versátil do AUTOSAR sendo estes, aspetos muito positivos no processo de desenvolvimento.

Uma aplicação que siga a arquitetura AUTOSAR é constituída por composições de componentes de *software*, possivelmente hierárquicas, designados por SW-Cs e que comunicam usando portas, definindo assim interfaces (Figura 2.12). As portas fornecem acesso a uma conexão ponto-a-ponto entre componentes. Essa conexão pode seguir modelos de comunicação padrão, como cliente-servidor ou recetor-emissor, para trocar dados ou invocar operações (Piper et al., 2012).

No entanto, é importante salientar que a visão do sistema, segundo a camada de aplicação, é diferente da implementação do sistema. Segundo a visão da aplicação, os SW-Cs estão diretamente ligados entre si através das suas respetivas portas e interfaces, mas esta visão não é real porque não

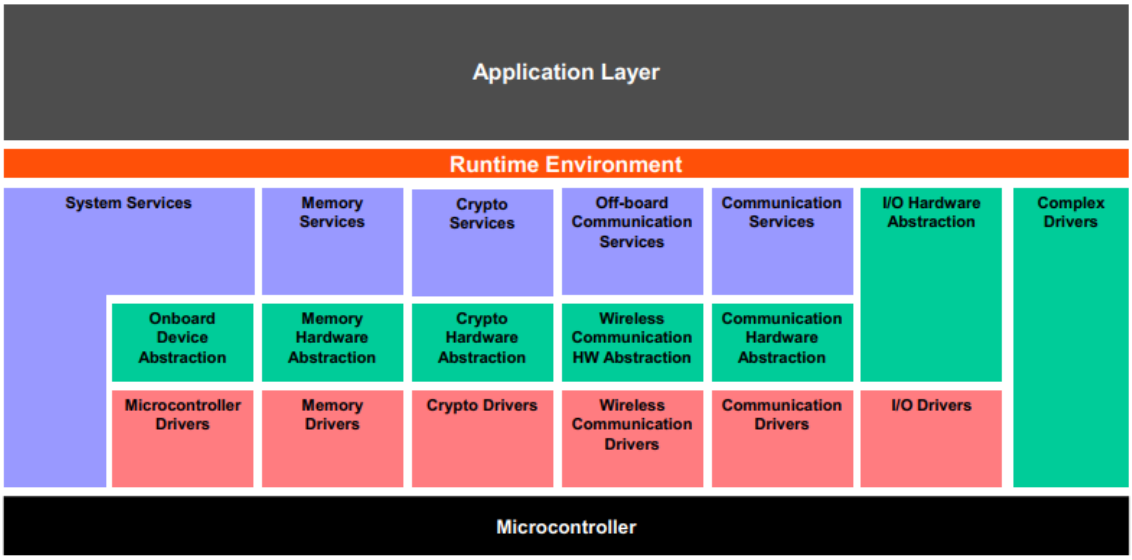


Figura 2.10: Arquitetura detalhada do AUTOSAR (Piper et al., 2012).

existe comunicação direta entra os SW-Cs na implementação. Em vez disso, cada SW-C invoca a API do RTE, que providência os serviços/primitivas para a comunicação entre componentes.

A camada RTE é responsável pela implementação da comunicação entre SW-Cs e faz a abstração do canal da comunicação real. Para enviar as mensagens o RTE usa os serviços fornecidos pela camada BSW, que é composta por sub-camadas permitindo fazer a abstração do *hardware*. A Figura 2.11 representa a abstração da comunicação entre os componentes A e B, contudo, o modo como é implementada pode resultar em três formas diferentes (Figura 2.12). Caso os componentes A e B residam no mesmo ECU a comunicação pode envolver apenas o RTE ou o RTE e o BSW. No caso de os componentes serem distribuídos, i.e., os componentes A e B estão em diferentes ECUs, então a comunicação utiliza a rede física, e.g. CAN.

2.3.2 Descrição funcional

A metodologia AUTOSAR descreve as etapas que devem ser executadas ao longo do desenvolvimento do sistema. Trata-se de *templates* para o desenvolvimento do modelo de aplicação. Estes *templates* não impõem uma ordem rigorosa nas atividades que devem ser realizadas. Os

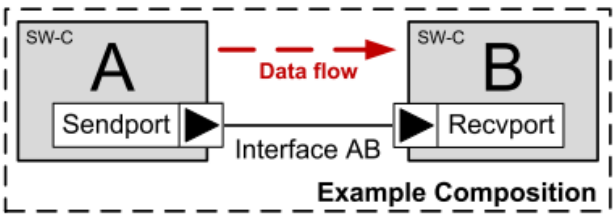


Figura 2.11: Comunicação entre dois SW-Cs com uma interface recetor-emissor (Piper et al., 2012).

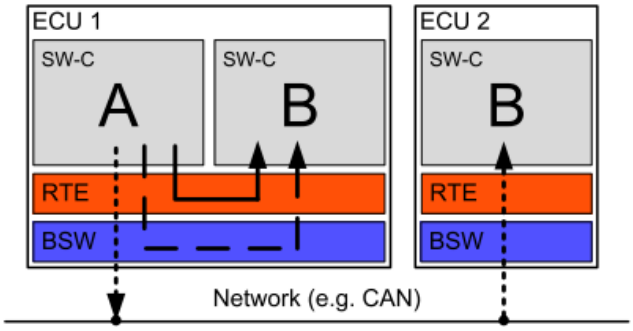


Figura 2.12: Modelo da comunicação entre dois SW-Cs (Piper et al., 2012).

templates do AUTOSAR não define quem, quando e onde é feito o desenvolvimento de *software*. Os procedimentos e a partilha das diferentes funções e/ou tarefas são acordados entre o OEM e os seus fornecedores, usando ferramentas capazes de manipular esses modelos. No primeiro passo, são especificadas as informações do sistema que vão ser usadas para a configuração inicial. Essas informações são descrições formais dos SW-Cs, recursos de *hardware* do ECU e topologia do sistema. A configuração do sistema tem em conta essas descrições e executa um mapeamento dos SW-Cs para uma ou mais ECUs. Com a utilização do RTE, o *software* da aplicação é independente do *hardware*.

Sempre que se utiliza a *framework* AUTOSAR é necessário ter em conta o *schema* que é utilizado, uma vez que normaliza os elementos, evitando, deste modo, diferentes significados ou termos sem sentido (e.g. codificação de caracteres e identificações). Ao utilizar o *schema* tipicamente expresso em linguagem XML é possível reduzir o esforço necessário no desenvolvimento da ferramenta porque estabelece a quantidade de *Tags* (ou elementos) possíveis de encontrar no documento (e.g. prefixos de *namespace* diferentes, codificação de caracteres, nomes de arquivos). Com o *schema* também são estabelecidas regras para a representação do documento XML, tais como:

- A extensão do documento tem do ser .arxml
- A codificação de caracteres das descrições XML tem de ser Unicode Transfomation Format (UTF)-8. Nenhuma outra codificação é permitida.
- A descrição AUTOSAR XML não deve começar com a UTF *Byte Order Mask*
- Declarar no início a versão do XML e a origem do documento: *root element* (*xsi:schemaLocation*)

Estes são exemplos de algumas das características que podem ser encontradas em documentos AUTOSAR e que são necessárias para uma correta interpretação pelas ferramentas. Os documentos AUTOSAR XML (ARXML) descrevem o sistema, ou parte dele, podendo ser designados como *system extract* / *system descrition* e o ECU *Extract* depende do nível de decomposição que é feita sobre o sistema.

- ***System Descrition*:** na descrição do sistema estão representados cinco elementos: onde estão presentes restrições na topologia, identificado o *software* e a comunicação usada, como também o seu mapeamento e possíveis restrições.
- ***System Extract*:** descreve uma visão específica do subsistema na descrição completa do sistema. O sistema não está totalmente decomposto e ainda contém composições;
- **ECU *Extract*:** é a informação da configuração do sistema (descrição) necessária para um ECU específico como, por exemplo, os sinais a que tem acesso;

Em termos técnicos não há diferenças entre *System Extract* e *System Descrition*. Contudo, é possível afirmar que as diferenças que existem estão relacionadas com questões de metodologias. Embora, quando se faz o mapeamento de um ECU, o *System Extract* é igual ao *System Descrition* menos os elementos que não são relevantes para o ECU em específico, como está representado na Figura 2.13.

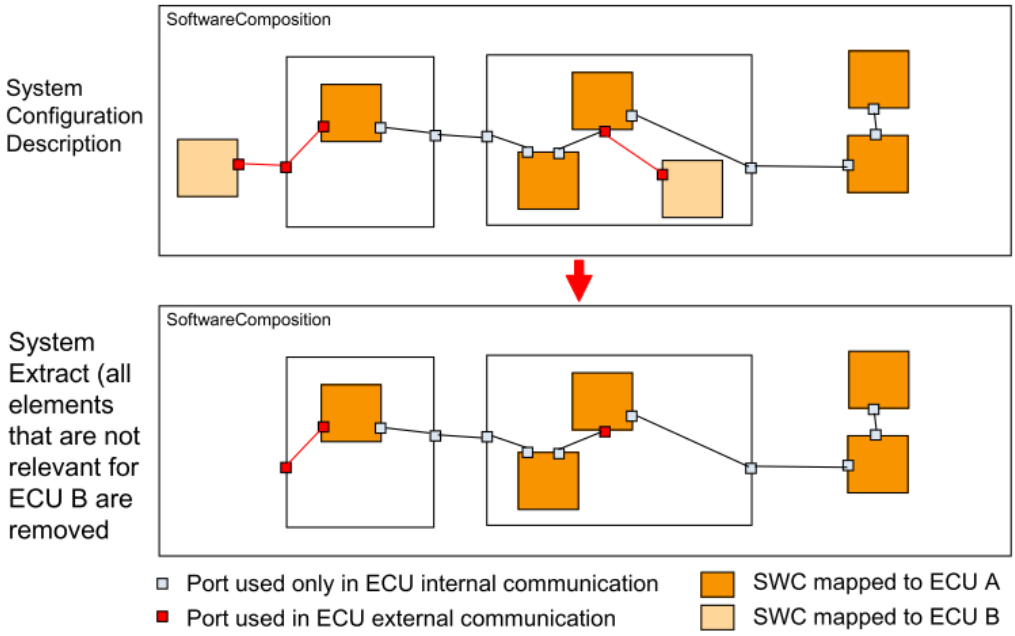


Figura 2.13: Arquitetura detalhada do AUTOSAR (Piper et al., 2012).

Estes tipos de documentos, que descrevem o sistema, estão em XML, mas quando é seguido a metodologia do AUTOSAR a extensão do documento é **.arxml*.

2.4 Linguagem XML

A linguagem XML é uma linguagem de marcação capaz de descrever diversos tipos de dados, cujo intuito é facilitar a partilha de informação. Este tipo de linguagem é muito utilizado na indústria para a partilha de dados porque o documento XML está organizado de forma hierárquica (estrutura definida pelo próprio usuário ou por um *schema*). Essencialmente, o documento XML

é uma árvore de elementos onde cada elemento pode ter um conjunto de atributos, elementos ou texto ou, por fim, uma mistura dos três. Os documentos XML seguem uma determinada estrutura definida pelo *schema*. Este *schema* tem uma grande importância no documento XML porque impõe um conjunto de regras que restringem a estruturação e o conteúdo do documento. Por outras palavras, o *schema* especifica a sintaxe e as semânticas presentes no ficheiro XML e o documento só é valido se, e só se, satisfazer essas restrições impostas pelo *schema* (Fialli, 2003).

Os documentos ARXML seguem uma determinada estrutura dependendo da versão do *schema* do AUTOSAR. Quando é criado um documento são seguidas as regras definidas pelo AUTOSAR, podendo haver algumas regras que coincidem com a elaboração do um documento XML normal, por exemplo:

- Todo documento XML, além da *Tag* introdutória, deve ter um único elemento (*Tag*) que sirva como raiz para todos os demais elementos do documento;
- O documento XML é *case sensitive*, ou seja, difere letras maiúsculas e minúsculas
- Todos elementos XML tem de ser iniciados e fechados

Após a elaboração dos documentos ARXML, ou XML, é necessário retirar a informação desses documentos. Para isso, existem ferramentas cujo intuito é fazer a análise sintática aos documentos de forma automática, conhecidas como ferramentas de *parsing*. Uma ferramenta de *parsing* é um *software* que lê o documento XML e torna a informação do ficheiro disponível para ser utilizada por aplicações e linguagens de programação. Existem duas abordagem diferentes para fazer a leitura do documento, a diferença encontra-se no modo como é feita a leitura do documento e como a informação é disponibilizada. A leitura/análise de um documento XML pode ser baseada em (Lam et al., 2008):

1. **Árvore**: Realiza a leitura do documento na totalidade e é construido em memória o modelo em árvore com a informação. Esta abordagem permite voltar para trás e para frente no documento XML que é lido. No entanto, o modelo criado normalmente é maior do que o documento original multiplicando a memória utilizada.
2. **Eventos:** A análise por eventos consiste na leitura do documento por secções e, sempre que encontrar um elemento novo surge um evento. Este modelo opera apenas de forma sequencial e em porções unitárias do documento.

2.5 Fluxo de desenvolvimento

O desenvolvimento de *software* passa por várias fases importantes, tais como planeamento, análise, design e implementação. Existem vários modelos para o desenvolvimento de *software*. Desses modelos os mais conhecidos são o *waterfall*, V-model e o modelo em espiral (*Spiral model*). Dentro destes modelos vão ser analisados os modelos *waterfall* o V-model e *Agile* porque são os modelos utilizados na BOSCH®.

2.5.1 Waterfall model

Modelo *waterfall* é o modelo antigo e bastante conhecido seguindo uma lógica de desenvolvimento sequencial (Figura 2.14), ou seja, uma sequência de estágios em que a saída de cada etapa se torna a entrada para o próximo (Balaji, 2012).

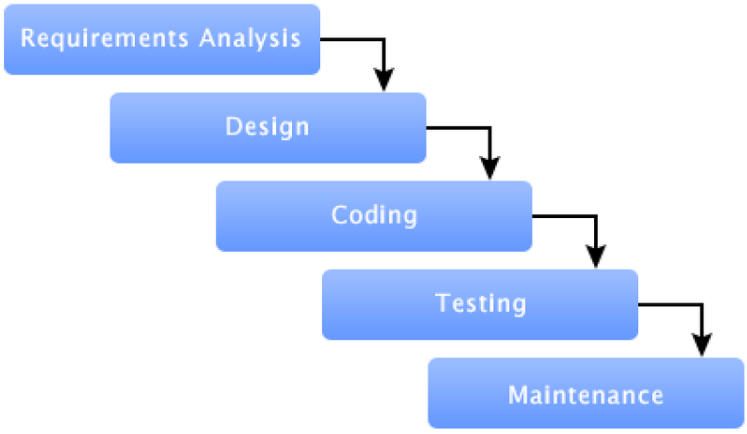


Figura 2.14: Modelo Waterfall (Calikus, 2016).

Ao utilizar este modelo é estabelecido um requisito claro no início do desenvolvimento, só se passa para a fase seguinte quando a fase anterior é completada dentro de um período de tempo estabelecido. A sua simplicidade e linearidade tornam este modelo fácil de implementar.

Por outro lado, uma das desvantagens deste modelo é que caso seja preciso alterar o requisito inicial, essa alteração não será implementada no processo de desenvolvimento atual. Outra das desvantagens, é este modelo estar mal estruturado porque na realidade muitos problemas relativos a uma fase surgem após a sua conclusão. Isto implica que os problemas relativos a uma fase podem não ser resolvidos completamente durante essa fase. Apesar das desvantagens, este modelo ainda é bastante utilizado para o desenvolvimento de *software* (Balaji, 2012).

2.5.2 V-model

Outro modelo muito utilizado por empresas é o V-model, sendo uma modificação do modelo *Waterfall*. Este modelo apresenta dois ramos: o ramo da verificação e o ramo da validação (Figura 2.15). Este processo de desenvolvimento é equilibrado e depende da verificação das etapas anteriores antes de prosseguir. O produto em cada fase precisa de ser verificado (ramo da verificação) e só mais tarde aprovado, após a verificação de todas as fases (ramo da validação) (Balaji, 2012). Este modelo trás diversas vantagens porque permite criar os testes e validar simultaneamente e caso haja alterações nos requisitos é possível alterar em qualquer fase. No entanto, este modelo tem a desvantagem de, se ocorrer alguma mudança ao longo do desenvolvimento, ser necessário atualizar os documentos dos requisitos e a documentação dos testes. Para além disso, não é recomendado para projetos curtos porque exige revisões em cada etapa.

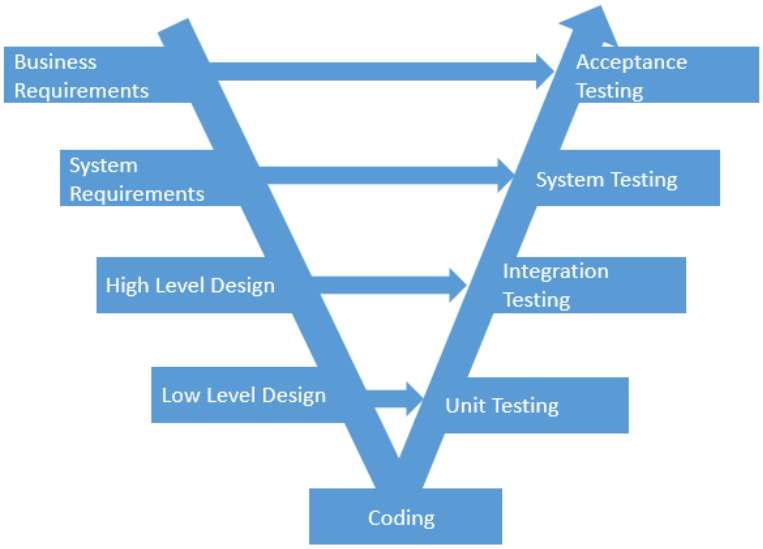


Figura 2.15: Ciclo de vida do V-model (Balaji, 2012)

2.5.3 Agile

Agile (Martin, 2002) é um termo que abrange várias metodologias de desenvolvimento de *software* de forma iterativa. O primeiro passo para aplicar estas metodologias é ter um âmbito e, subsequentemente, definido o projeto (também designado de *Epic*). Tendo este projeto definido são elaboradas pequenas e simples descrições de funções/características do projeto, narradas na primeira pessoa. Geralmente, estas descrições são feitas pelo utilizador ou cliente do sistema, conhecidas como *User Stories*. A partir destas *User Stories* são derivadas as tarefas a realizar e os responsáveis por elas (Figura 2.16).

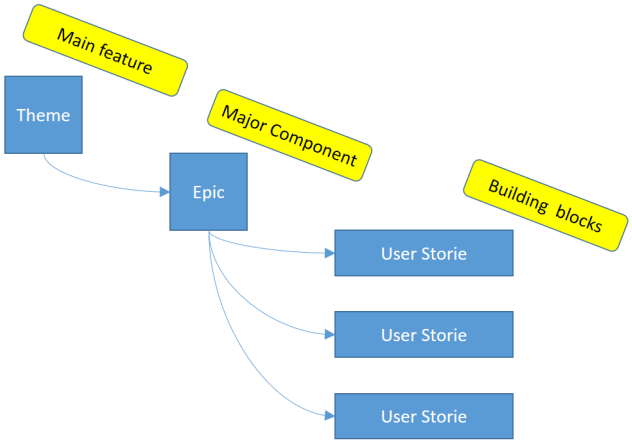


Figura 2.16: Linha cronologia de um sistema seguindo *Agile*[9].

[9]"Craft Blog"URL https://blog.craft.io/2017/04/23/writing-epic-user-stories/

2.6 Ferramentas de Teste

Num ECU *Extract* existe um conjunto de restrições e funcionalidade que o sistema tem de cumprir (requisitos). Após a especificação do sistema é muito importante validar estas características e funcionalidades. Para isso são necessários testes que obrigam o sistema a ter determinado comportamento. Contudo, antes de executar os testes é necessário elaborar os cenários em que os testes vão correr, quais são os objetivos e as condições necessárias para passar nos testes. Aqui entra o processo de RE que faz a documentação dos casos de teste (*Test Case Specifications*), onde são especificados os objetivos do testes, as condições necessárias para os realizar, os conjuntos de passos a ser executados e os critérios para poder-se afirmar que o sistema passou nos testes (Regnell et al., 1995).

2.6.1 CANoe

A ferramenta *CANoe* desenvolvida pela empresa VECTOR®(Piper et al., 2012) é particularmente relevante para a fase de teste. Trata-se de uma ferramenta comercial que fornece um ambiente de simulação e avaliação para aplicações automóveis (Figura 2.17). A ferramenta *CANoe* permite testar e analisar uma rede de ECUs ou apenas uma ECU individualmente. Esta ferramenta tem sido usada por OEMs e fornecedores há mais de 20 anos.

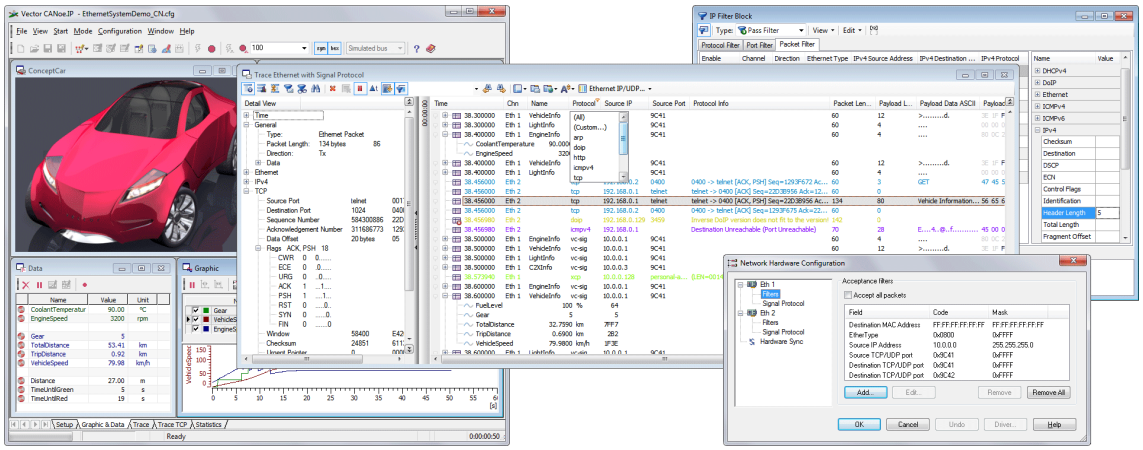


Figura 2.17: Ferramenta CANoe[10].

Esta ferramenta é usado pela BOSCH® para realizar os testes que validam as funcionalidades dos ECU *Extract*. O *CANoe* permite vários cenários onde podem ser testadas funcionalidades e comportamentos de forma independente ou em conjunto. O projetista é o responsável por definir os cenários e os testes que pretende executar (Zhou et al., 2008). A ferramenta apenas permite dividir o processo em 3 fases, que auxiliam o projetista no desenvolvimento do sistema e para a implementação do mesmo (Figura 2.18):

[10]"Vector:CANoe"URL https://vector.com/vi_canoe_ethernet_en.html

1. **Análises de requisitos e projeto do sistema de rede**: O utilizador, responsável pelo projeto, distribui a funcionalidade geral do sistema entre diferentes nodos da rede. Isso inclui definir mensagens e selecionar a taxa de transmissão de dados.

2. **Implementação dos componentes com a simulação do barramento:** Após a conclusão da primeira fase, um dos nodos criados na fase anterior é substituído por um nodo real e inserido na rede. Assim é possível simular o barramento e verificar o comportamento do nodo real. Isto pode ser feito para todos os nodos, um de cada vez.

3. **Integração de todo o sistema:** Nesta última fase de desenvolvimento só existem nodos reais na rede sendo *CANoe* utilizada como gerador/analisador de tráfego na rede. Assim, é possível verificar a reação dos nodos a cada mensagem que é introduzida no barramento.

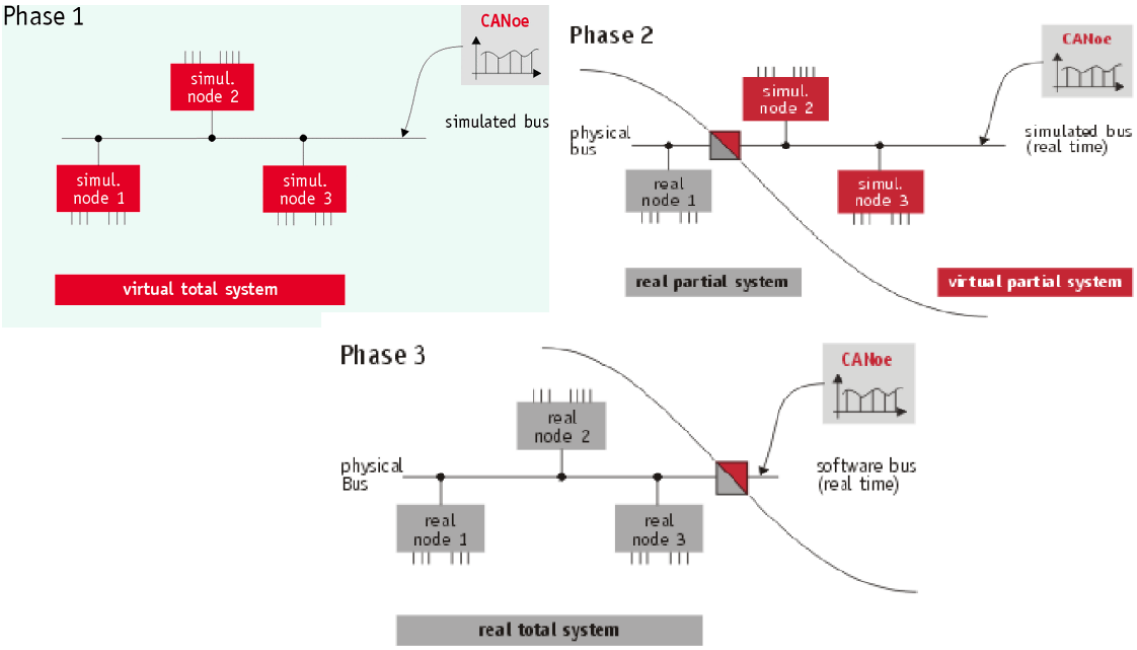


Figura 2.18: Divisão do processo de desenvolvimento usando CANoe (Hvanth et al., 2012).

Durante o processo de desenvolvimento esta ferramenta pode ser usada para integrar um protocolo de comunicação virtual, um protocolo físico ou para fazer a simulação do protocolo no sistema. Após o desenvolvimento, a ferramenta *CANoe* pode ser usada para analisar a rede física permitindo assim, obter valores estatísticos da rede (Hvanth et al., 2012). Em suma, a ferramenta permite verificar o comportamento, a distribuição funcional e integração do sistema.

2.7 Resumo do Capítulo

Neste capítulo foi descrito o desenvolvimento dos sistemas de controlo na industria automóvel, a importância de RE no desenvolvimento desses sistemas e foi feito um levantamento dos protocolos de comunicação utilizados na comunicação entre ECU. Foram focados os protocolos

FlaxRay, CAN, *Ethernet* e LIN apresentado as suas características, as vantagens e as desvantagens na sua utilização.

Foi também apresentado detalhadamente a *framework* AUTOSAR, a sua relevância para a industria e a sua estruturação. Por fim, foram apresentados algumas metodologias de trabalho no desenvolvimento de *software* neste contexto, mais especificamente, o fluxo de desenvolvimento e ferramentas de teste relacionadas com o tema desta dissertação.

Capítulo 3

Análise Inicial do Projeto

Neste capítulo serão discriminados todos procedimentos necessários à elaboração do projeto apresentado na presente tese. Não só será feita a descrição do projeto e o que é pretendido como também, a metodologia de trabalho utilizada juntamente com o planeamento. Será também elaborado um levantamento das linguagens de programação e das ferramentas de *parsing* disponíveis. Por fim, será fundamentado a escolha da linguagem de programação e da ferramenta de *parisng*.

3.1 Definição do problema

Hoje em dia na indústria automóvel, mais especificamente na empresa BOSCH®, o processo para criar os requisitos e as descrições passo-a-passo dos casos de teste para um ECU *Extract* é feito manualmente. Isto implica que sempre que a BOSCH® recebe um novo ECU *Extract* de um cliente é necessário analisar o documente, elaborar a lista de requisitos do sistema e depois criar os casos e teste para validar esses requisitos. Esta lista de requisitos descreve de forma objetiva o comportamentos e características do sistema. Sempre que um cliente acrescentar novas funcionalidades, ou apenas faça alterações no ECU *Extract*, isto implica analisar novamente o documento, reescrever os requisitos e os casos de teste para o novo ECU *Extract*.

A BOSCH® colabora com vários clientes, de quem normalmente recebe uma nova versão de um ECU *Extract* a cada três meses, ou seja, de três em três meses são atualizados manualmente os documentos sobre um determinado sistema. Esta abordagem ttraz duas grandes desvantagens que afetam a produtividade de uma empresa: i) o elevado tempo dispensado na definição e validação dos requisitos de um sistema; e ii) o aumento da probabilidade de erros humanos. Desta forma, para ultrapassar estas desvantagens, é crucial automatizar o processo de definição e criação de requisitos de um sistema bem como a descrição dos testes a serem utilizados na validação dos mesmos.

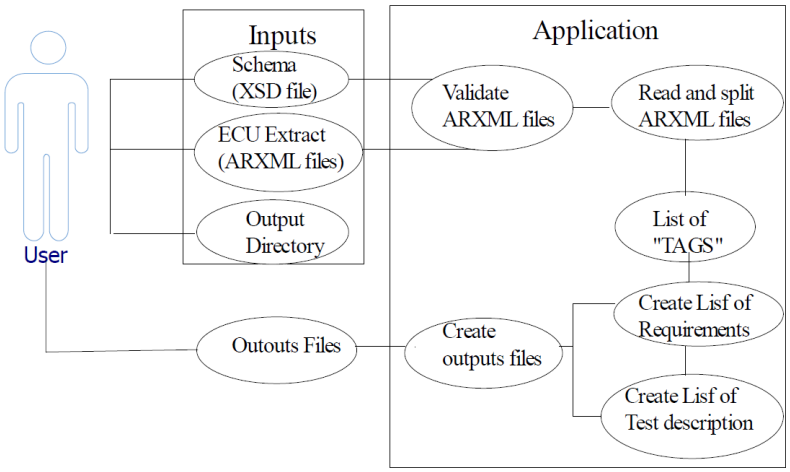


Figura 3.1: Diagrama de caso de uso.

3.2 Solução pretendida

Nesta dissertação pretende-se desenvolver uma solução que possa integrar os sistemas de RE da BOSCH®, cuja funcionalidade é produzir os requisitos de sistema e de *software* relacionados com um ECU *Extract* bem como, gerar as descrições dos testes para a validação dos mesmos.

Foram feitas algumas imposições por parte da BOSCH® no desenho da aplicação que se prendiam, maioritariamente, com o formato e estrutura dos dados de *input* e *output* suportados pela aplicação. Desta forma, com base nas discussões com a equipa de desenvolvimento da BOSCH® foi elaborada uma lista de requisitos funcionais (RF) e não funcionais (RNF) apresentada na Tabela 3.1 a serem cumpridos pela aplicação. Com base nesta tabela, elaborou-se um diagrama de caso de uso da aplicação pretendida (Figura 3.1). Assim sendo, o utilizador necessita de definir os *inputs* da aplicação, sendo eles: os documentos com extensão ARXML do ECU *Extract*, os documentos com extensão *XSD* referentes ao esquemático do AUTOSAR pelo qual o ECU *Extract* se guiou, e o diretório para o *output*. Inicialmente, a aplicação irá avaliar o documento ARXML verificando se são cumpridas as normas imposta pelo esquemático. Após esta validação, a aplicação irá fazer a leitura do documento ARXML e criar objetos das classes correspondentes às *tags* encontradas bem como estabelecer as ligações entre os próprios objetos. Com esta lista de objetos, irá ser gerada a lista de requisitos e descrições passo-a-passo dos testes de validação desses requisitos. Posteriormente, esta informação será disponibilizada ao utilizador em formado *Excel* no diretório escolhido inicialmente pelo utilizador.

Na proposta feita pela empresa, foi determinado que o desenvolvimento da aplicação deveria priorizar os requisitos dos protocolos de comunicação, mais especificamente, os protocolos CAN e *Ethernet*. Por isso, a solução proposta foca-se nesses dois protocolos de comunicação, estabelecendo os parâmetros para os quais devem ser criados requisitos e as respetivas descrições. Contudo, durante o desenvolvimento é tido em conta a possibilidade e a facilidade de estender a aplicação, permitindo que futuramente seja possível avaliar outros aspetos para além dos protocolos de comunicação mencionados anteriormente.

Tabela 3.1: Requisitos da aplicação.

Identificador	Descrição
RF1	A aplicação deverá ser executada no sistema operativo *Windows*
RF2	A aplicação deverá permitir configurar a versão do AUTOSAR a utilizar
RF3	A aplicação deverá trabalhar com a versão 4.3 do AUTOSAR
RF4	A aplicação deverá permitir configurar o caminho dos ficheiros ARXML
RF5	A aplicação deverá permitir configurar o caminho dos ficheiros de *output*
RF6	A aplicação deverá ser capaz de ler a especificação técnica do ECU *Extract*
RF7	A aplicação deverá verificar se o ECU *Extract* segue o esquemático utilizado
RF8	A aplicação dever ser capaz de selecionar/repartir a informação do ECU *Extract*
RF9	A aplicação deverá conseguir derivar requisitos através do ECU *Extract*
RF10	A aplicação deverá produzir um ficheiro de *output* com a lista de requisitos
RF11	A aplicação deve criar os requisitos para os protocolos de comunicação
RNF1	A lista de requisitos deverá ter a seguinte estrutura: *Primary Text* | *Artifact type* | *Status* | *Review state* | *Requeriments SourceID* | *Verification type*
RNF2	Os requisitos criados para CAN deverão incluir as características do protocolo

Após estabelecer os requisitos e o caso de uso foi feito um planeamento para o desenvolvimento da aplicação passando por diversas etapas. Para estabelecer este planeamento foi necessário conhecer primeiro as metodologias de trabalho implementadas na empresa e como é feita a sincronização entre os elementos da equipa de desenvolvimento.

3.3 Metodologias de trabalho

No inicio das atividades na BOSCH®, foi realizado o processo de integração com duração de duas semanas. Neste processo foi apresentada a empresa, as equipas de trabalho das diferentes áreas presentes na BOSCH® de Braga, e as metodologias de trabalhos utilizadas. Dentro das

Figura 3.2: Exemplo genérico de um *Kanban Board*[1].

metodologias foi apresentado o modelo *Agile* juntamente com o *Kanban* e o *Scrum*. Estes métodos de trabalho foram aplicados no decorrer do desenvolvimento da aplicação, facilitando o apoio e a sincronização dos elementos da equipa contribuindo para um aumento na eficiência do trabalho.

3.3.1 Agile

Dentro da metodologia *Agile*, apresentada na Secção 2.5.3, existe a metodologia *kanban board* (Hammarberg e Sunden, 2014) que é uma das ferramentas usadas na implementação do *Kanban* na gestão do trabalho, ao nível pessoal e/ou empresarial.

3.3.1.1 *Kanban board*

O *kanban board* permite visualizar a organização do trabalho e o seu fluxo. Com a utilização do *kanban board* é possível visualizar o estado de cada tarefa, bem como quem é o responsável por cada uma. Na figura 3.2 está representado um *Kanban Board* genérico, onde estão presentes três colunas, que representam o estado das tarefas e indiretamente o estado do projeto.

- ***To Do*:** Na primeira coluna estão as listas de tarefas a realizar. Estas tarefas estão associadas aos *User Stories* sendo também estabelecido quem é o responsável pela execução das mesmas.
- ***Work in Progress (WiP)*:** Lista de tarefas que estão a ser realizadas no momento bem como, a identificação do responsável por executar cada tarefa podendo ser uma pessoa ou uma equipa. Nesta fase os elementos podem associar breves comentários sobre o ponto de situação ou problemas que estão a ter.
- ***Done*:** Nesta coluna são colocadas todas as tarefas que já foram concluídas e o tempo que demoraram até á sua conclusão. Devem também ser colocados os problemas encontrados durante a execução das tarefas pela equipa ou pessoa encarregada de as executar.

[1] "You Will Do Better- FIXE -Change Management"URL http://www.youwilldobetter.com/2012/05/treinamento-kanban-como-ferramenta-na-gestao-de-projetos-2/

Este quadro permite que todos os elementos da equipa tenham conhecimento do ponto da situação relativamente à execução das tarefas sendo esta uma das principais vantagens desta metodologia.

3.3.1.2 *Scrum*

Também dentro do *Agile*, existe uma *framework* chamado *Scrum* (Schwaber, 2004), utilizada no desenvolvimento de *software*, seguindo o conceito de *User Storie*. Esta *framework* tem como objetivo planear as tarefas em *sprints*. Para cada *sprint* é feito uma simples descrição, narradas na primeira pessoa, do que é pretendido no final do *sprint*, ou seja, a cada sprint está associada uma *User Storie*, podendo ter uma duração de uma a quatro semanas. Os *User Stories* são derivados de um *Epic* (projeto). Normalmente, um *Epic* é constituído por quatro *sprints* (quatro *User Stories*). A junção destes quatro *sprints* é chamada *Baseline* que corresponde ao período disponível para alcançar o objetivo estabelecido no *Epic*.

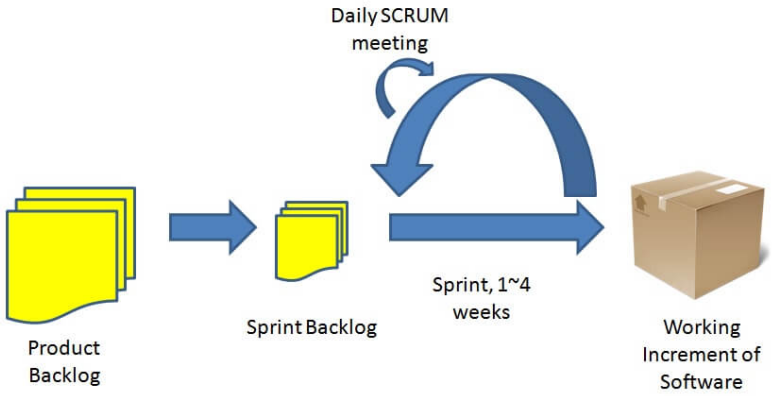


Figura 3.3: Fluxo do *SCRUM*[2].

Na figura 3.3 é representado o fluxo de trabalho do *Scrum*, tendo inicio no *Epic* (*Product Backlog*), que é dividido em um *User Story* para cada *sprint* (*Sprint Backlog*) e termina o ciclo quando o produto está no estado final previsto no *Epic*. Durante os *sprints* são feitas, diariamente, reuniões curtas (*Daily meetings*) com todos os elementos da equipa para fazer uma revisão do que foi feito no dia anterior. Estas reuniões têm como objetivo relatar os pontos altos do dia (*highlights*), se houve algo problema (*blockers*) e, por fim, definir os objetivos para o dia. Isto permite que todos os elementos estejam sincronizados com as tarefas que estão a ser realizadas. No fim de cada *sprint* é feita a revisão (*review*) e a retrospetiva, onde é feito uma balanço individual sobre as tarefas que foram realizadas e é feito o planeamento do próximo *sprint*.

[2]"Done before Brekky"URL http://donebeforebrekky.com/scrum-magic-book-review/

Tabela 3.2: Planeamento do *base line*.

Sprint(Date)	User Story	Tasks (Hours Spent)
1 (8Mar-5Apr)	As a user, I want a list of tags present in the ECU Extract **So that means (DoD):** - The application provides to the user a dialogBox to choose the ARXML files - The application validates the ARXML files - The application produces a list of TAGs presented in the files	- Integration weeks (100h) - Parsing an ARXML file (20h) - Parsing Framework and schema AUTOSAR (30h)
2 (5Apr-16Apr)	As a user, I want a list of all elements present in the ECU extract **So that means (DoD):** - The application produces an Excel file with the list of all elements - The application validates the ARXML files	- Parsing the ARXML files-2^{nd} Iteration (50h) - Validate the ARXML files using the schematic of AUTOSAR 4.3(8h)
3 (17Apr-30Apr)	As a user, I want a list of requirements for the elements present in the ECU extract **So that means (DoD):** - The application produces the requirements for each element of the ECU Extract	- Relation between requirement and AR-package (5h) - Types of requirements (10h) - List of requirements (50h)
4 (Apr30-May14)	As a user, I want a list of test descriptions to the elements present in ECU extract **So that means (DoD):** - From a list of requirements, the application produces a description step-by-step of the test to validate the requirements	- Validation of the requirements' templates (13h) - Associate a test to a group of requirements (2h) - Produce the list of test case Specification (40h) - Define the format of the test description (2h)

3.4 Planeamento

O planeamento para esta dissertação teve em conta as metodologias de trabalho apresentadas na secção anterior que estão implementadas na empresa BOSCH®. Assim sendo, o principal objetivo desta dissertação e, consequentemente, do *Epic*, é elaborar uma aplicação que gera automaticamente requisitos e descrições de caso de teste a partir de um ECU *Extract*. Deste *Epic* foram derivados *User Stories* para uma *base line* com duração de dois messes.

Na Tabela 3.2 estão presentes as *user stories* para cada *sprint* do *baseline* que derivaram do *Epic*. Cada *sprint* teve uma duração de duas semanas em que a complexidade das tarefas foi aumentando, de forma gradual, focando-se cada vez mais na função principal da aplicação. Começando com uma lista de tarefas relativamente simples e abrangentes (tarefas do primeiro *sprint*) até às tarefas para executar a função de gerar requisitos e descrever casos de testes (tarefa do ultimo *sprint*). Para cada tarefa também foi feita uma estimativa do tempo necessário para a realizar, sendo necessário no final de cada *sprint* fazer a correção do tempo realmente gasto.

Foram estimadas 330 horas para a criação da aplicação. Neste planeamento não foram consideradas as horas para a escrita do documento da dissertação nem as reuniões com a equipa de desenvolvimento sobre outros projetos. Estas tarefas foram executas em paralelo ao planeamento presente na Tabela 3.2.

3.5 Levantamento de Ferramentas

Com o planeamento estabelecido, foi dado inicio ao desenvolvimento tendo sido feito, inicialmente, um levantamento das linguagens de programação e das *frameworks* de *parsing* existentes. Este levantamento enquadra-se na tarefa *Parsing Framework and schema AUTOSAR* do primeiro *sprint*.

As *frameworks* de *parsing* a utilizar dependem diretamente da linguagem de programação usada. Nesta fase de levantamento de ferramentas a serem utilizadas foram estabelecidos três aspetos relevantes para a pesquisa: i) quais as linguagens de programação que se enquadravam com as necessidades da aplicação; ii) de que forma é feita a análise e a recolha de informação de um documento XML (parsing) e iii) como é feita a validação de um documento XML de acordo com um esquemático AUTOSAR.

3.5.1 Linguagem de programação

Tendo em conta as características do problema, constatou-se que a linguagem *C* é a menos adequada devido à complexidade da aplicação. Deste modo, optou-se por uma linguagem orientada a objetos. Assim, dentro das linguagens orientadas a objetos existem as linguagens *C++*, *C#*, *Python* e *Java* como opções. O facto de a todas elas serem linguagens orientadas a objetos facilita, à partida, a estruturação da aplicação uma vez que satisfaz as suas necessidades e apresenta uma relação direta com a linguagem Unified Modeling Language (UML) utilizada para projetar a arquitetura da aplicação. Considerando estas quatro linguagens de programação e tendo em conta o conhecimento adquirido ao longo da minha formação, não se optou pelas linguagens *Python* e *C#* por falta de qualquer contacto prévio. Dentro das restantes, *C++* e *Java*, foi feito um levantamento das ferramentas de *parsing* existentes para cada uma. Tendo em conta as características e os recursos que cada uma disponibilizava optou-se por usar a linguagem *Java*. Como esta linguagem é multiplataforma permite correr a aplicação em qualquer o sistema operativo. Esta característica permite satisfazer o requisito RF1 (Aplicação ser executada no sistema operativo *Windows*) da lista de requisitos da aplicação presentes na Tabela 3.1. Outra vantagem desta escolha é a facilidade de criação de interfaces para a interação como utilizador.

3.5.2 Ferramentas de *parsing*

Após a escolha da linguagem de programação foi realizada uma pesquisa mais aprofundada das ferramentas que a linguagem *Java* disponibiliza para o *parsing* de um documento XML (Haw e Rao, 2007), resultando num levantamento de 4 ferramentas de *parsing*:

- ***DOM Parser/Builder*:** É uma fermenta do *Java* que fornece uma interface ao programador para aceder à informação de um documento XML. Trata-se de uma ferramenta capaz de analisar todo o documento XML de uma só vez, traduzindo o seu conteúdo numa árvore estabelecendo relações hierárquicas, onde na base da árvore está o elemento *root* do documento. São utilizados os termos de *Parent* ("Pai"), *Child* ("Filho") e *Sibling* ("Irmão") para estabelecer a relação hierárquica entre os elementos na árvore (Figura 3.4).

- ***Simple API for XML (SAX) Parser*:** Esta ferramenta faz a análise do documento de forma sequencial tendo em conta uma metodologia baseada em eventos. Em vez de executar a

[3] "Parsing XML using DOM parser in Java"URL http://www.topjavatutorial.com/java/java-programs/parsing-xml-dom-java/

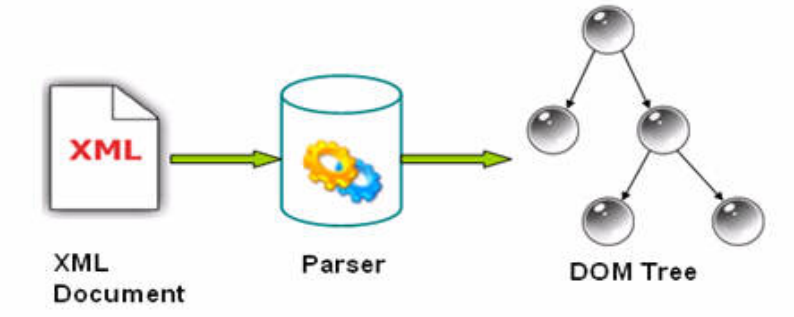


Figura 3.4: DOM *parser* [3].

leitura de todo o documento de uma só vez, o SAX só reporta um elemento de cada vez, o que implica usar menos memória comparativamente ao DOM. Neste caso, apenas é obtido um elemento de forma independente sem estabelecer ligação com os seus antecessores.

- ***StAX Reader/Write***: Esta ferramenta faz uma análise misturando o modo em árvore (DOM) com o modo por eventos (SAX). Esta ferramenta permite fazer uma análise parcial do documento permitindo, também, obter informações dos antecessores do elemento. De uma forma metafórica o *StAX* introduz um cursor "livre"no documento. O contrário acontece no SAX, onde o cursor só pode seguir num sentido.

- ***JAXB*:** Esta ferramenta permite ao programador aceder e analisar um documento XML sem ter a necessidade de conhecer a estrutura do documento. Isto é possível devido ao facto de a ferramenta apenas necessitar de ter o esquemático correspondente ao documento sendo a análise feita baseada no esquemático. Uma das características desta ferramenta é traduzir os elementos XML em objetos, preenchendo todos os seus atributos. O JAXB também permite fazer o processo inverso, passar de uma lista de objetos para um documento XML(Figura 3.5).

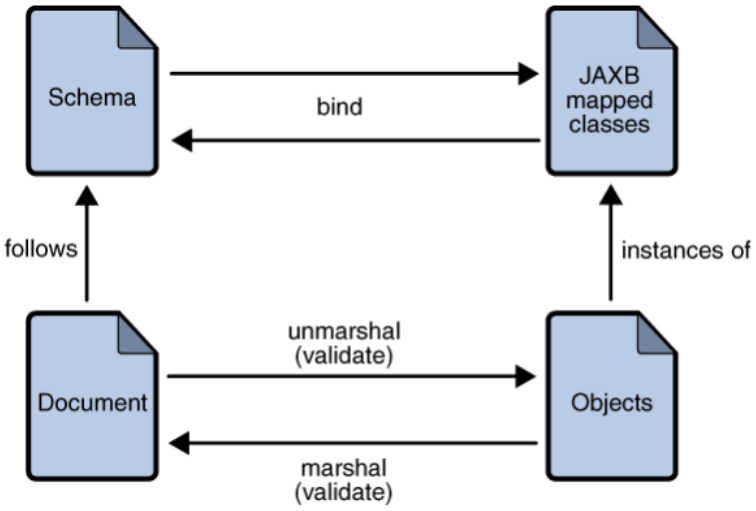


Figura 3.5: Arquitetura do JAXB[4].

Devido à complexidade do documento ARXML e às características da ferramenta DOM, não foi possível utilizar esta ferramenta para fazer a análise do documento. Como a estruturação feita pelo DOM é baseada em árvore, não é possível ter um elemento como *Parent* noutro nível hierárquico porque cada elemento apenas pode ter um *Parent*. No entanto, no documento ARXML é possível encontrar esta situação pois é possível ter o mesmo elemento como *Parent* e *Child*.

Uma das necessidades que existe no AUTOSAR é saber o caminho percorrido até chegar a um elemento, sendo para isso necessário saber todos os seus antecedentes. A ferramenta SAX não permite tal procedimento tendo sido descartada. Desta forma, sobraram as ferramentas *StAX Reader/Write* e o JAXB. Devido ao facto do ECU *Extract* seguir uma estrutura especificada pelo esquemático do AUTOSAR, é necessário seguir o esquemático para fazer o *parsing*.

Ao contrário do *StAX Reader/Write*, o JAXB utiliza o esquemático do documento XML como referência para fazer a desserialização dos elementos. Além disso, o JAXB também permite validar o documento ARXML verificando se foram cumpridas as regras impostas pelo esquemático. Esta característica permite validar o requisito RF6 (verificar se o ECU *Extract* está bem estruturado) da Tabela 3.1.

3.6 Resumo do Capítulo

Neste capitulo foi descrito detalhadamente o problema e a solução pretendida. Aqui foram descritos os requisitos funcionais e não funcionais da aplicação bem como um caso de uso. De seguida, foram introduzidas as metodologias de trabalho usadas na empresa BOSCH® e que serviram de base para planeamento do desenvolvimento desta dissertação. Por ultimo, foi realizado um levantamento das linguagens e das ferramentas de *parsing* para o desenvolvimento da aplicação, tendo sido apresentados os motivos que levaram à escolha da linguagem *Java* e da ferramenta JAXB.

[4] "The Java web Services Turorial "URL http://fizpmkf1.fic.uni.lodz.pl/ podlaski/java/javaeetutorial5/doc/JAXB2.html

Capítulo 4

Gerador Automático de requisitos e descrições de casos de teste

Este capítulo apresenta a primeira abordagem à arquitetura da aplicação, utilizando a linguagem UML, mais especificamente, o diagrama de classes e o diagrama de sequência para a descrever. Depois será feita uma análise da arquitetura para detetar problemas e aspetos a melhorar. Como consequência dessa análise foi necessário fazer uma reestruturação da arquitetura. Sendo apresentado a arquitetura final da aplicação que preenche as lacunas da arquitetura inicial (protótipo). Já com a arquitetura final é demonstrado o processo da aplicação a gerar automaticamente os requisitos e o os casos de teste.

4.1 Protótipo

A utilização da linguagem UML facilita o desenvolvimento da aplicação porque permite organizar, planear e visualizar a estrutura da aplicação, especialmente no caso de programação orientada a objetos. Além disso, por se tratar de uma linguagem padrão, é facilmente traduzida para a linguagem *Java*. Por estes motivos, é utilizado o diagrama de classes para fazer uma representação estática da aplicação, indicando as classes que constituem a aplicação, bem como as relações entre elas. Por outro lado, o diagrama de sequência é usado para fazer uma descrição dinâmica da aplicação, descrevendo a sequência de processos para um caso de uso.

Na primeira abordagem à arquitetura da aplicação o diagrama de classes traduzia em classes as entidades principais da aplicação e as relações entre elas. As relações seguem a sequência lógica das funções da aplicação, ou seja, a primeira coisa que a aplicação deve fazer é permitir ao utilizador escolher os documentos de entrada e o local de destino para os documentos de saída. Após a ação do utilizador é necessário validar e analisar cada documento ARXML. A análise consiste em reconhecer cada *Tag*, recolher os seus valores e derivar os requisitos. Depois de ter os requisitos gerados é necessário elaborar os casos de teste para validar os mesmos. Nos casos de teste é descrita a sequência de passos a realizar para executar os testes (Figura 4.1). Assim sendo, esta sequência de ações deu origem a uma arquitetura dividida em cinco classes.

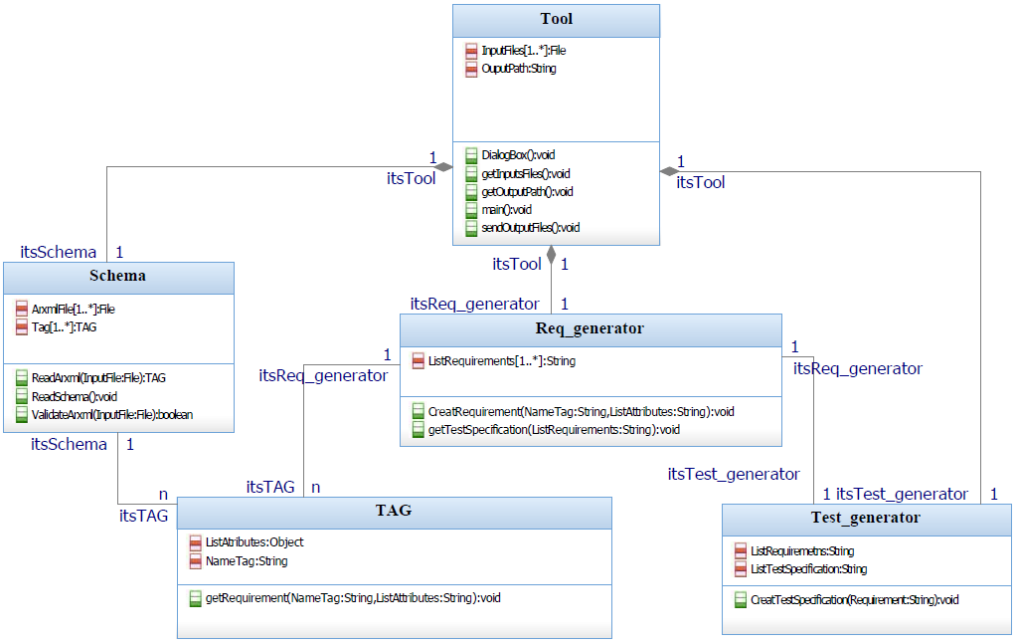


Figura 4.1: Diagrama de classes.

- **Classe *Tool*:** Nesta classe está presente a função responsável pela interação com o utilizador, nomeadamente a seleção dos *inputs*: os documentos XML (esquemático e o ECU *Extract*) e o caminho de destino para os documentos gerados. Depois desta função, é dado início ao processo de geração de requisitos e testes.

- **Classe *Schema*:** Após ter todos os *inputs* é necessário fazer a verificação dos documentos e de seguida a análise do conteúdo de cada um. Estas funções são executadas nos métodos da classe *Schema*. A validação é feita no método *ValidateArxml*, verificando se os documentos cumprem as regras impostas pelo esquemático. Se não houver problemas com os documentos são recolhidas as informações utilizando o método *ReadArxml*, criando objetos da classe TAG e preenchendo os respetivos parâmetros.

- **Classe *TAG*:** Esta classe trata-se de uma representação abstrata de cada elemento. Os diferentes tipos de elementos são definidos pelo esquemático do AUTOSAR. No fim de cada elemento encontrado no documento vão ser derivados os requisitos. Contudo, esta função não é atribuída a esta classe mas sim, à classe *Req_generator*.

- **Classe *Req_generator*:** Nesta classe vai ser elaborada a lista de requisitos para cada elemento, onde cada elemento possui os seus próprios requisitos. Para isso é necessário utilizar um método que esteja responsável por gerar os requisitos de acordo com as normas impostas pelo processo RE.

- **Classe *Test_generator*:** Após a criação de todos os requisitos, vão ser criados os casos de teste para a validação dos mesmos, descrevendo o cenário e a sequência de passos para

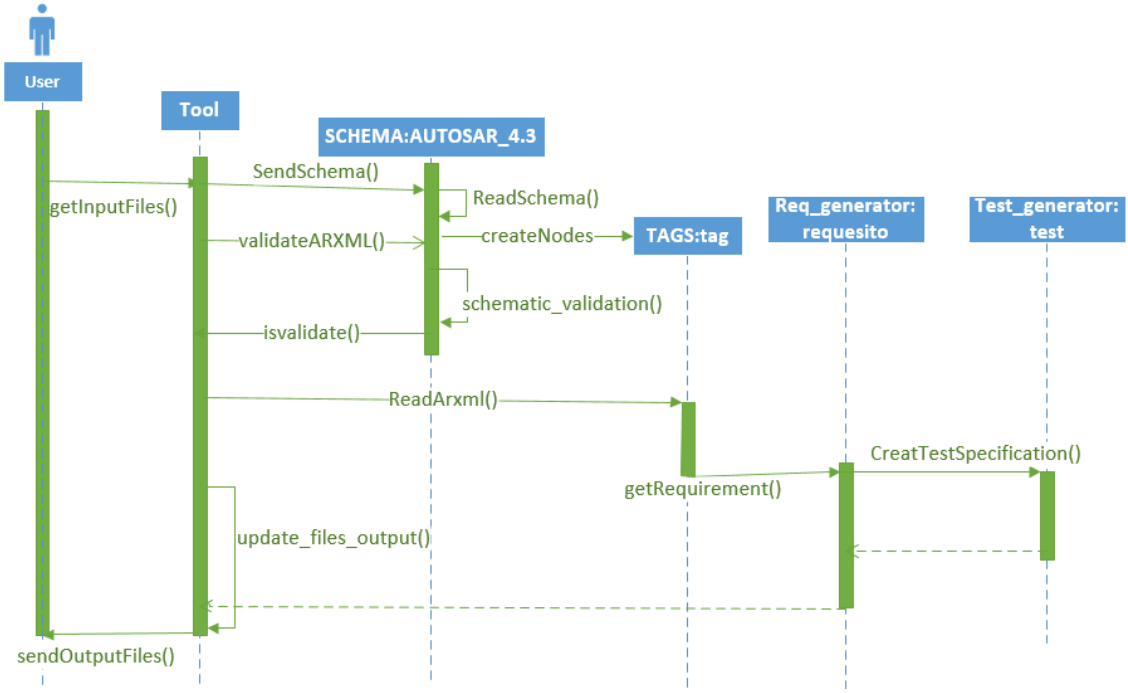


Figura 4.2: Diagrama de Sequência.

realizar o teste. Esta função é atribuída à classe *Test_generator* que vai ter também métodos para gerar um documento com as descrições dos casos de teste.

Seguindo o diagrama de classes apresentado anteriormente, foi elaborado um diagrama de sequência para projetar e descrever os processos e as relações entre classes que estão envolvidas na execução da aplicação (Figura 4.2).

O diagrama de sequencia tem início com as ações do utilizador escolhendo o documento ARXML e o esquemático a ser utilizado. De seguida, no objeto da classe *Schema* é avaliado o documento ARXML. Só depois de passar na verificação do documento é feito o *parsing* do ECU *Extract*. O *parsing* consiste em encontrar as *Tags* (elementos presentes no ECU *Extract*) e recolher os respetivos atributos. Por fim, no objeto do tipo *Req_generator*, vão ser gerados requisitos utilizando os valores dos atributos dos elementos e as respetivas descrições dos casos de teste. Estas informações são guardadas em documentos *Excel* no diretório escolhido pelo utilizador. Desta forma é concluído o processo de geração de requisitos e casos de teste a partir de um ECU *Extract*.

4.1.1 Discussão da arquitetura

Após elaborar a arquitetura inicial (Figura 4.1) e ao longo do desenvolvimento da aplicação foram detetadas lacunas na arquitetura. Estas lacunas estavam relacionadas com o modo como era realizada a recolha da informação, a geração dos requisitos e a relação entre os requisitos e as descrições de casos de teste gerados.

Segundo a arquitetura do protótipo (arquitetura inicial) todas as *Tags* deveriam ser analisadas, contudo, foi determinado pela equipa de desenvolvimento que nem todas as *Tags* eram relevantes de momento. Neste sentido não havia necessidade de as analisar uma vez que estas seriam irrelevantes para os requisitos pretendidos. Durante o desenvolvimento foram detetadas lacunas no processo de criação e escrita dos requisitos relacionados. O primeiro problema encontrado na criação dos requisitos foi que as *Tags* não são independentes, ou seja, existem casos em que os atributos necessários para gerar um requisito estão em várias *Tags* não permitindo a associação direta entre uma *Tag* e um requisito. Outro problema foi o facto de que os requisitos gerados teriam de seguir certas normas que não estavam a ser consideradas na arquitetura do protótipo.

Por fim, outro dos problemas encontrados na arquitetura inicial foram as relações entre os requisitos e os casos de teste que indiretamente também estão relacionadas com a falta de processos padrão. O problema consistia em elaborar os passos necessários para fazer a descrição do teste para cada tipo de requisito. Essa descrição implica criar um cenário, a lista de passos a realizar e os critérios de validação para cada passo onde é possível existir situações em que o mesmo caso de teste possa ser usado para validar requisitos diferentes.

4.2 Arquitetura final

Como resultado da análise da arquitetura do protótipo foi necessário refazer a arquitetura da aplicação com o intuito de solucionar os problemas encontrados na primeira versão. Nesta secção explicamos a abordagem utilizada para chegar à arquitetura final da aplicação e como foram solucionados os problemas referidos na secção anterior.

4.2.1 Diagrama de classes

Na reestruturação da arquitetura foram separados quatro principais funções: i) receber os *inputs*, compreende-los e escrever os *ouputs*; ii) filtrar os dados dos *inputs*; iii) criar a lista de requisitos; iv) criar a lista de casos de testes.

Após definir estas funções principais, foi detalhado o comportamento para cada função, determinando as sub-funções e as dependências. Isto levou à arquitetura representada pelo diagrama de classes da Figura 4.3, com 10 classes das quais:

- **Classe *App*:** Nesta Classe está presente a função *main*, onde são pedidos ao utilizador os documentos ARXML e o local dos *outputs*. Após o utilizador definir estes parâmetros, a aplicação vai dar início à leitura dos documentos ARXML, à criação dos requisitos e às descrições de testes através dos métodos das classe *AutosarIO*, *RequirementGenerator* e da classe *TestGenerator* respetivamente.

- **Classe *AutosarIO*:** A função mais básica da aplicação, mas das mais importantes, é fazer o *parsing* do documento ARXML. É neste documento que está a especificação técnica do sistema. A classe *AutosarIO* possui os métodos responsáveis por fazer a validação e a

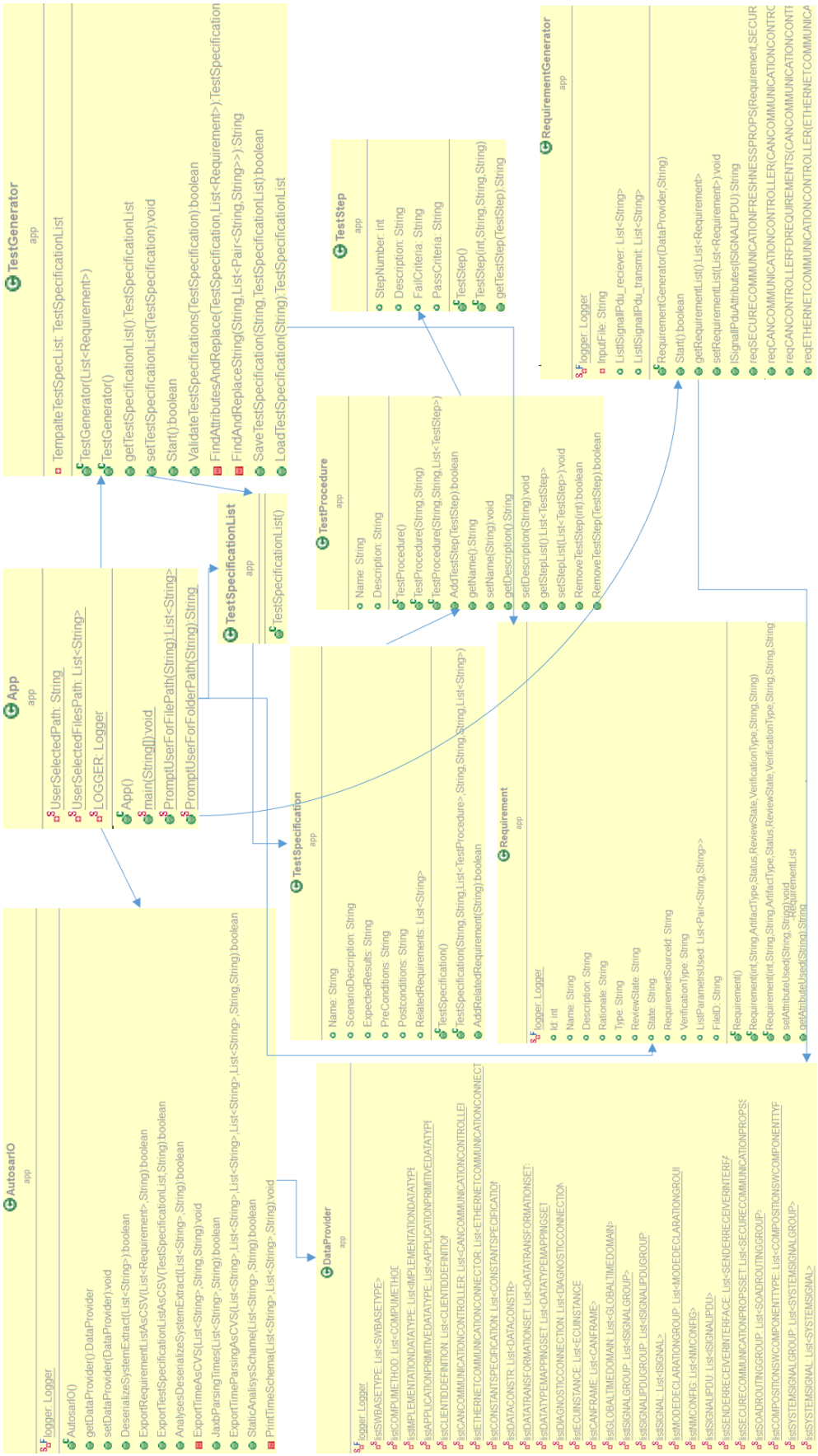


Figura 4.3: Diagrama de Classes Final.

análise do documento e, para isso, a classe tem um método que utiliza a ferramenta JAXB para fazer o *parsing*. Esta ferramenta utiliza o método *Unmarshaller* para traduzir o texto XML para objetos, preenchendo os respetivos atributos.

- **Classe *DataProvider*:** Esta classe contém a lista dos elementos (*Tags* do AUTOSAR) mais relevantes para os requisitos. Esta filtragem foi feita após discussão com a equipa de desenvolvimento e seguiu uma lista de elementos determinados pelo cliente tendo em conta os tipos requisitos pretendidos.

- **Classe *RequirementGenerator*:** Uma das funcionalidades desta aplicação é criar requisitos a partir da especificação técnica presente no ECU *extract*. A classe *RequirementGenerator* tem os métodos responsáveis por elaborar essa lista de requisitos. Para executar esta tarefa é necessário extrair os dados do ECU *extract* (extração feita pelos métodos da classe *AautosarIO* e a filtragem com os métodos da classe *DataProvider*). Estes dados vão ser utilizados no método *GenerateRequirements* para criar e preencher a estrutura do requisito (estrutura definida pela classe *Requirement*). No final, é esperado ter uma lista de objetos do tipo *Requirement*, ou seja, uma lista de requisitos relacionados com ECU *extract* ;

- **Classe *Requirement*:** Esta classe define a estrutura de um requisito tendo também os métodos responsáveis pelo preenchimento dos atributos. Assim sendo um requisito é definido por um conjunto de atributos (Tabela 4.1) tais como: o estado do requisito, o nome do ECU *extract*, a descrição e o tipo de requisito. Este atributos são obrigatórios para criar um requisito.

Tabela 4.1: Atributos da Classe *Requirement*.

Atributo	**Tipo**	**Descrição**
Id	int	Número de identificação do requisito
Name	*String*	Nome requisito
Description	*String*	Breve descrição do requisito
Rationale	*String*	Breve descrição do motivo para criar este requisito
ArtifactType	*Enum*	Tipo de requisito (Requisito de *Software* ou de Sistema)
ReviewState	*Enum*	Se o requisito já passou pela fase de revisão (*Pending* / Finished / Working)
Status	*Enum*	Estado do requisito (*New* / *Approve* / *Not Aprove*)
RequirementSourceId	*String*	Identificação do requisito *template*
ListParametrsUsed	*List «Pair {String , String}»*	Lista de atributos utilizados no requisito com o respetivos valores
FileID	*String*	Nome do ECU *Extract*

- **Classe *TestGenerator*:** A quarta função principal desta aplicação é fazer a descrição dos casos teste para validar os requisitos. A classe *TestGenerator* é a responsável por essa função relacionando os requisitos com os *templates* dos testes. Para criar um caso de teste é necessária criar um cenário onde se pode testar o comportamento relacionado com os requisitos. A descrição do teste (*TestSpecification*) está divida em dois etapas: i) descrever o processo do teste (*TestProcedure*); ii) descrever os passos do processo (*TestStep*).

- **Classe *TestSpecificationList*:** Esta classe foi criada devido à necessidade de ter um elemento *Root* no documento XML com todos os *templates* dos testes, que vão ser transformados em

test Case Specifications. Esta classe só tem como atributo uma lista de objetos do tipo *TestSpecification*. O processo de gerar casos de teste a partir do *template* irá ser explicado mais à frente no sub-capitulo 4.2.4

- **Classe *TestSpecification*:** O objeto desta classe vai representar uma especificação para um caso de teste, tendo preenchidos os atributos apresentados na Tabela 4.2. Na especificação do caso de teste estão referenciados os requisitos que deram origem à especificação e respetiva descrição do processo para realizar o teste (objeto do tipo *TestProcedure*).

Tabela 4.2: Atributos da classe *TestSpecification*.

Atributo	**Tipo**	**Descrição**
ScenarioDescription	*String*	Descrição do cenário de teste
Name	*String*	Nome requisito
Procedure	*List<TestProcedure>*	Lista de processos
ExpectedResults	*String*	Resultado esperado na realização do teste
PreConditions	*String*	Condições que são necessárias de se verificar antes de realizar o teste
Postconditions	*String*	Condição que deve ser garantida depois do teste
Status	*Enum*	Estado do teste (*New* / *Approve* / *Not Aprove*)
RelatedRequirements	*List<String>*	Lista dos tipos de requisitos que associados ao teste (*templates Id*)

- **Classe *TestProcedure*:** Esta classe é responsável por fazer a descrição do processo, ou seja, fazer a descrição sequencial dos passos necessários para realizar o teste. Esta descrição representa uma lista de objetos da classe *TestStep*.

- **Classe *TestStep*:** Para cada teste existe um conjuntos de passos a realizar com uma determinada ordem. Recordando a metodologia *Waterfall model* (Sub-secção 2.5.1), só é possível passar para o passo seguinte se, e só se, verificar um determinada condição. Esta classe é responsável por fazer a descrição do "*step*"e estabelecer o resultado esperado para poder passar para o "*step*"seguinte.

4.2.2 Diagrama de sequência

Na elaboração de um diagrama de sequência foi representado um caso de uso onde a aplicação deriva apenas um requisito do documento e apenas um caso de teste. No entanto, é possível derivar mais requisitos e ter mais casos de teste sendo que, para simplificar a explicação da arquitetura, foi representado um caso de uso básico (Figura 4.4).

Após o utilizador selecionar o ECU *Extract*, é dado o início à análise do documento invocando o método *AutosarIO.DeserializeSystemExtract* que tem como parâmetro o documento, ou documentos, ARXML. Durante este método são encontrados os elementos que vão fazer parte do objeto do tipo *DataProvider*. No final do método *DeserializeSystemExtract*, da classe *AutosarIO*, são retornadas as listas de elementos (atributos da classe *DataProvider*).

De seguida, começa o processo de gerar os requisitos invocando o método *Start* da classe *RequirementGenerator*. Dentro deste método são criados objetos do tipo *Requirement*, mas só são

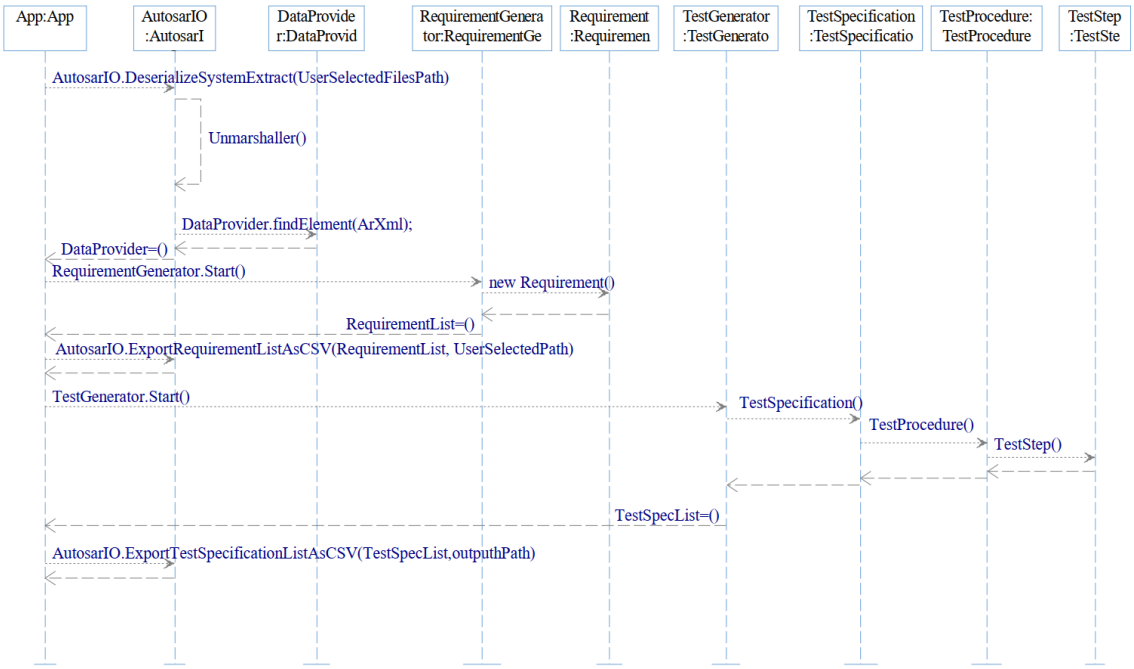


Figura 4.4: Diagrama de sequencia da arquitetura final.

criados os requisitos quando todos os parâmetros usados no *template* são encontrados e substituídos. Depois de serem gerados todos os requisitos possíveis, estes são tratados pelo objeto *App* onde a partir do método *ExportRequirementListAsCSV*, da classe *AutosarIO*, é criado um ficheiro *Excel* com todos os requisitos e guardado no diretório escolhido pelo utilizador.

Após a geração dos requisitos, dá-se início ao processo para gerar os casos de teste invocando o método *Start*, da classe *TestGenerator*. Dentro deste método vão ser gerados os casos de teste e para isso é criado um objeto do tipo *TestSpecification* para cada teste. Nesses objetos vão estar descritos os processos juntamente com a lista de passos que especificam cada caso de teste. No final do processo é enviado para o objeto *App* a lista de todos os casos de teste. Esta lista vai ser passada para um ficheiro *Excel* através do método *ExportTestSpecificationListAsCSV* da classe *AutosarIO*. Assim, no final temos dois ficheiros *Excel*, um com todos os requisitos e outro com as descrições dos casos de teste.

4.2.3 Gerador automático de requisitos

Como o processo de gerar requisitos e casos de teste era feito manualmente não existiam requisitos *standard*. Os requisitos eram criados apenas seguindo um conjunto de normas definidas internamente. No entanto, o modo como é escrito o requisito pode ser diferente, dependendo do colaborador que está responsável por esta tarefa. Isto também implica que sempre que são gerados requisitos é necessário fazer uma revisão por outro colaborador. Por isso, foi estabelecido durante a dissertação, que havia a necessidade de criar *templates* para os requisitos. Os *templates* tem um formato *standard* com "*placeholders*"para os valores das especificações técnicas provenientes do ECU *Extract*.

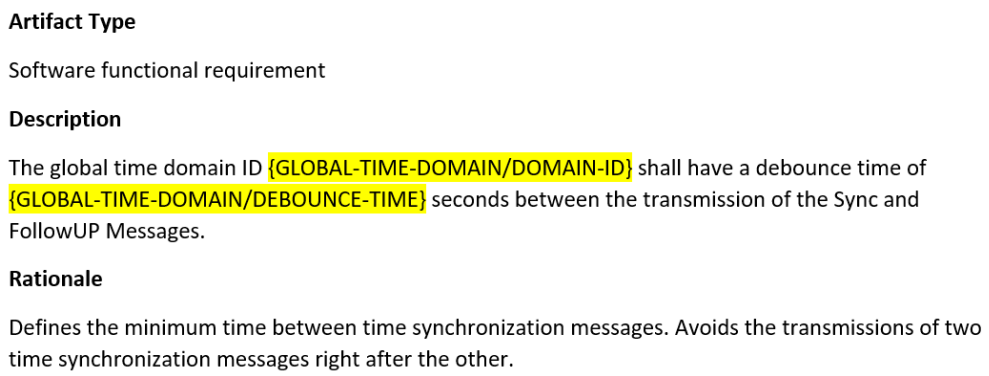


Figura 4.5: *Template* SWFRS_GTS_00001.1.

Para a aplicação gerar automaticamente os requisitos foi estabelecida uma estrutura que divide os requisitos por *features* e dentro de cada *feature* são divididos em requisitos de Sistema e de *Software*. Nestes padrões estão presentes as *Tags* que vão fazer parte do requisito. Em alguns casos apenas a *Tag* não é suficiente para obter o valor, é necessário ter o caminho, ou seja, o nome dos antecessores até chegar ao elemento e dentro desse elemento o nome do atributo.

Na Figura 4.5 está presente um *template* para um requisito de *software* relacionado com a *feature* do *Global Time Domains*, relativo ao protocolo Precision Time Protocol (PTP). No *template* são necessários os campos que definem o tipo de requisito, a descrição do genérica requisito e o motivo pelo qual é gerado (*Rationale*). Na descrição é onde vão estar as *Tags* que depois serão substituídas pelo respetivo valor.

Na Figura 4.6 são apresentados todos os *templates* gerados com o número de identificação, a que *feature* estão associados e a relação entre os requisitos de Sistema (SYSFRS) e de *software* (SWFRS). A árvore não está completa porque o processo de gerar os requisitos *template* ainda está a decorrer. Até ao momento, os *templates* que foram criados e aprovados estão relacionados com as *features Global Time Domains* e com o *Communications Interfaces*. Os restantes estão em fase de revisão.

4.2.4 Gerador automático de descrições de casos de teste

Para gerar os casos de teste que permitem validar os requisitos gerados foi estabelecida uma estrutura do caso teste (Tabela 4.2), onde estão presentes os passos do processo de teste, a identificação do teste, o motivo para o realizar e um campo onde estão identificados os tipos de requisitos que o teste valida (identificação do *template* do requisito). Assim, para implementar o processo de gerar automaticamente descrições de casos de teste são utilizados *templates* que definem o caso de teste. Na Figura 4.7 está presente um *template* para um caso de teste que é usado para validar os requisitos que derivam de dois tipos diferentes (*template* SWFRS_GTS_00001.12 ou *template* SWFRS_GTS_00002.12).

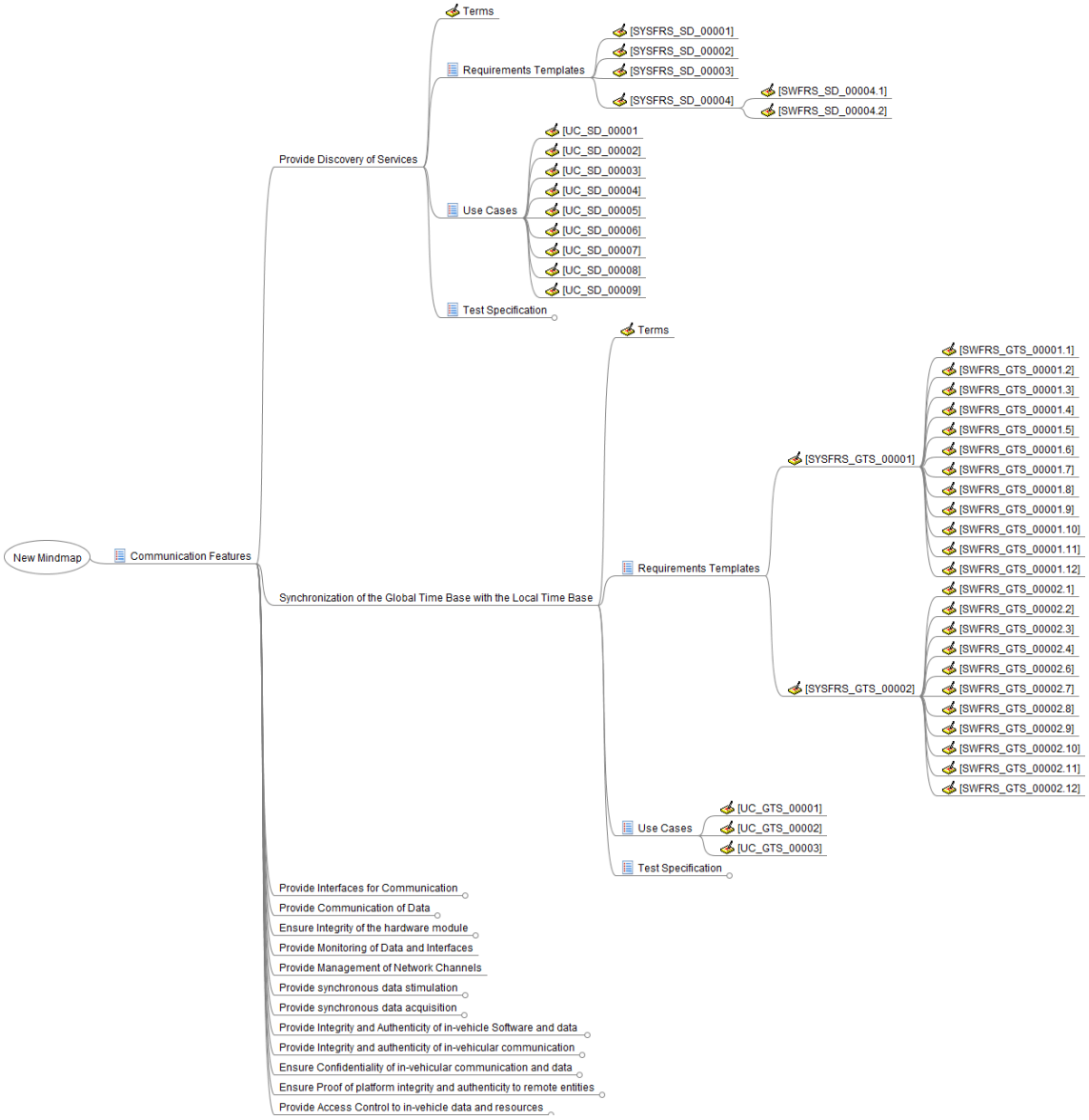
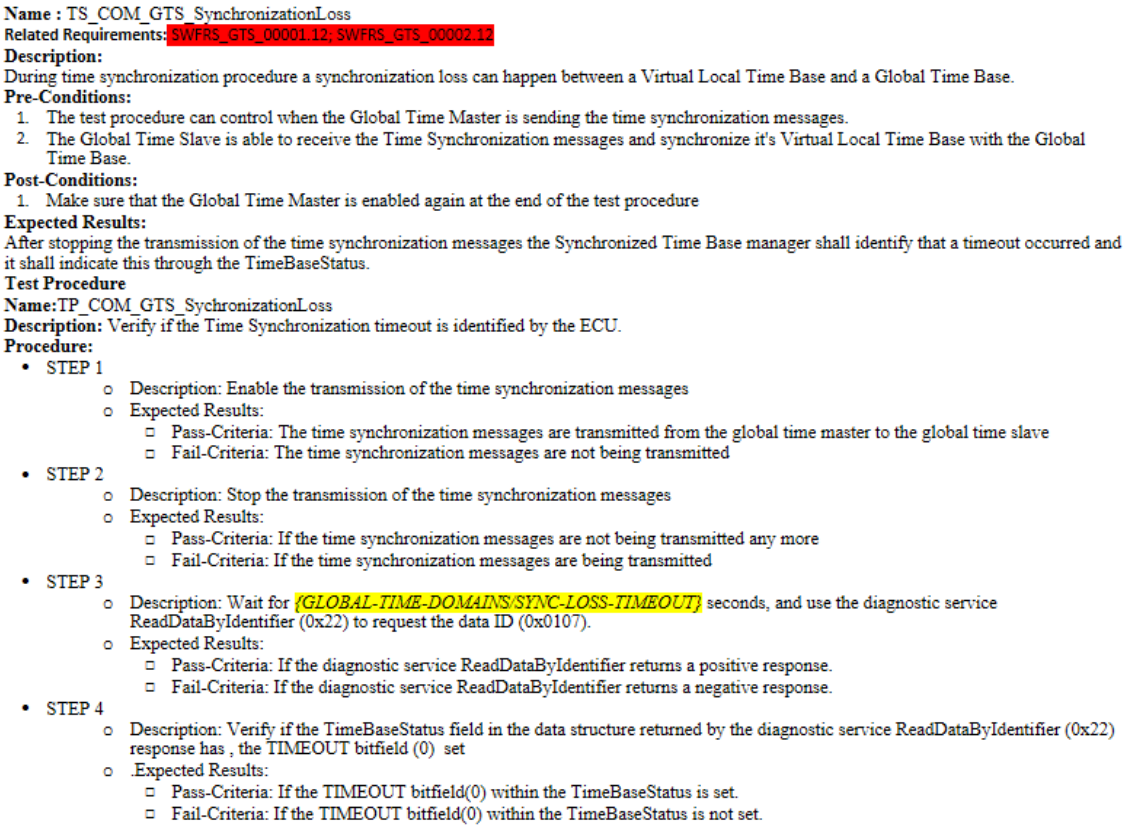


Figura 4.6: Estruturação dos requisitos.



Name : TS_COM_GTS_SynchronizationLoss
Related Requirements: SWFRS_GTS_00001.12; SWFRS_GTS_00002.12
Description:
During time synchronization procedure a synchronization loss can happen between a Virtual Local Time Base and a Global Time Base.
Pre-Conditions:
1. The test procedure can control when the Global Time Master is sending the time synchronization messages.
2. The Global Time Slave is able to receive the Time Synchronization messages and synchronize it's Virtual Local Time Base with the Global Time Base.

Post-Conditions:
1. Make sure that the Global Time Master is enabled again at the end of the test procedure

Expected Results:
After stopping the transmission of the time synchronization messages the Synchronized Time Base manager shall identify that a timeout occurred and it shall indicate this through the TimeBaseStatus.
Test Procedure
Name:TP_COM_GTS_SychronizationLoss
Description: Verify if the Time Synchronization timeout is identified by the ECU.
Procedure:
- STEP 1
  - Description: Enable the transmission of the time synchronization messages
  - Expected Results:
    - Pass-Criteria: The time synchronization messages are transmitted from the global time master to the global time slave
    - Fail-Criteria: The time synchronization messages are not being transmitted
- STEP 2
  - Description: Stop the transmission of the time synchronization messages
  - Expected Results:
    - Pass-Criteria: If the time synchronization messages are not being transmitted any more
    - Fail-Criteria: If the time synchronization messages are being transmitted
- STEP 3
  - Description: Wait for *{GLOBAL-TIME-DOMAINS/SYNC-LOSS-TIMEOUT}* seconds, and use the diagnostic service ReadDataByIdentifier (0x22) to request the data ID (0x0107).
  - Expected Results:
    - Pass-Criteria: If the diagnostic service ReadDataByIdentifier returns a positive response.
    - Fail-Criteria: If the diagnostic service ReadDataByIdentifier returns a negative response.
- STEP 4
  - Description: Verify if the TimeBaseStatus field in the data structure returned by the diagnostic service ReadDataByIdentifier (0x22) response has , the TIMEOUT bitfield (0) set
  - .Expected Results:
    - Pass-Criteria: If the TIMEOUT bitfield(0) within the TimeBaseStatus is set.
    - Fail-Criteria: If the TIMEOUT bitfield(0) within the TimeBaseStatus is not set.

Figura 4.7: Exemplo de um template para o caso de teste.

Os *templates* estão definidos num ficheiro XML, permitindo, de forma simples, adicionar novas descrições de caso de teste. A aplicação sempre que é executada vai ler este ficheiro e apenas gera testes para os requisitos que tenham algum *template* associado.

Após feita a relação entre os requisitos e os casos de testes, a dificuldade encontrada estava na ligação entre os atributos utilizados nos requisitos com os atributos necessários nos testes. Para solucionar este problema foi acrescentada nos requisitos uma variável com todos os atributos (*Tags*) usados e os respetivos valores. Para ter esta relação entre as *Tags* e os valores usados uma lista do tipo *Pair*. Com esta lista de atributos (lista de *Pair*) ao gerar a descrição do caso de teste basta fazer a substituição direta das *Tags* (atributos) pelos valores.

Um *Pair*, em *JAVA*, é um tipo de atributo que tem dois campos, *Key* e *Value*, estes dois campos estão associados, ou seja, para cada *Key* existe apenas um *Value*. Esta variável permite relacionar o atributo da *Tag* (*Key*) com o seu valor (*Value*). Cada requisito vai ter uma lista de atributos deste tipo, representado todas as *Tags* utilizadas e os respetivos valores. Esta lista vai permitir fazer a substituição das *Tags* presentes na descrição do processo do teste porque basta invocar o método *replace* (*replace(pair.getKey(), pair.getValue())*), para realizar a substituição do atributo (*pair.getKey*) pelo respetivo valor (*pair.getValue*).

4.3 Resumo do Capítulo

Este capítulo apresentou uma solução para a aplicação desejada, começando com uma proposta inicial de arquitetura que foi depois reformulada até à respetiva versão final. Recorrendo à linguagem UML para representar a arquitetura usando diagramas de classes e de sequência. Desta forma, foram apresentadas quais eram as limitações da versão inicial e como foram resolvidas.

Relativamente à arquitetura final foi apresentado o modo como foi implementado o processo de gerar automaticamente os requisitos e os casos de teste. Estes processos passam pelo uso de *templates* que estabelecem o elo de ligação entre os requisitos e os casos de teste. Por fim, foram ainda mostrados exemplos de *templates* e o modo como é feita a substituição dos atributos nos *templates* pelos os respetivos valores.

Capítulo 5

Implementação e resultados

Neste capítulo são discriminados todos procedimentos da aplicação, tais como, a interação com o utilizador, a fase de validação, análise do ECU *Extract* e os estudos realizados para o tempo que a aplicação demora a analisar os documentos, utilizando diferentes abordagens na leitura. Para além disso, são apresentados os requisitos e as descrições de casos de teste (*outputs*) gerados automaticamente pela aplicação.

5.1 Interação com o utilizador

Antes de iniciar o *parsing* do ARXML são pedidos ao utilizado os documentos de *input* e o diretório de destino para guardar os *outputs* gerados. Para fazer a interação com o utilizador é usada a biblioteca *JFileChooser*[1]. Esta biblioteca permite criar uma interface (*DialogBox*) para a interação com o utilizador representada na Figura 5.1 (por questões de confidencialidade foram omitidos dados). Esta *DialogBox* permite ao utilizador escolher o/s documento/s ARXML passando a aplicação para a fase seguinte. Podem ser escolhidos um ou mais ECU *Extract* porque a aplicação tem a capacidade de analisar múltiplos *inputs*. Nesta fase, o utilizador pode sair da aplicação carregando na opção "*Cancel*".

Após o utilizador escolher os *inputs*, surge uma nova *DialogBox* para que seja definido o diretório onde se pretende que os documentos gerados sejam guardados (requisito RF4 da Tabela 3.1). Concluída esta interação, a aplicação vai dar início à análise, de forma sequencial, dos documentos ARXML e sendo depois gerados os requisitos e os casos de teste.

5.2 *Parsing* do ECU *Extract*

Recordando a sub-secção 3.5.2, a ferramenta utilizada para o *parsing* foi o JAXB. Esta ferramenta permite criar classes a partir do esquemático introduzido pelo utilizador e estabelecer a ligação entre os elementos presentes no ARXML e objetos.

[1] https://docs.oracle.com/javase/7/docs/api/javax/swing/JFileChooser.html

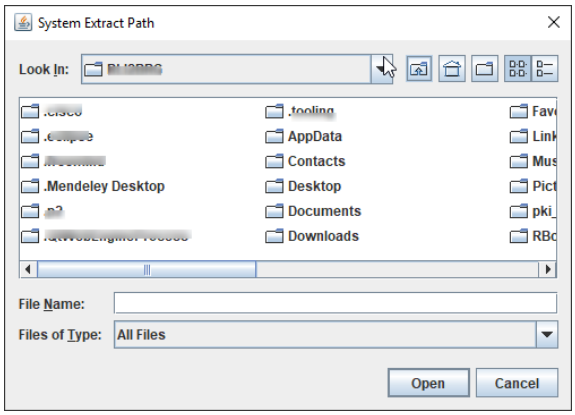


Figura 5.1: Interação gráfica com o utilizador(*DiagolBox*)

Durante o *parsing* do documento ARXML é feita uma validação do documento, ou seja, se são cumpridas as normas impostas pelo esquemático (RF7 da Tabela 3.1). Na aplicação são avaliados dois aspetos: i) se o documento não tem nenhuma *Tag* desconhecida, e ii) se o conteúdo da *Tag* é o correto, como por exemplo, se dentro da *Tag X* só são possíveis encontrar elementos do tipo A e/ou do tipo B, não é permitido encontrar elementos do tipo C. Se o ECU *Extract* passar nestes dois critérios, no final, existe um objeto do tipo AUTOSAR (elemento *root* do documento ARXML) que contém todos objetos criados a partir do ECU *Extract* com os seus respetivos atributos e métodos.

O método da ferramenta JAXB invocado para fazer *parsing* é o método *Unmarshaller*. Este método permite analisar os *inputs* (ECU *Extract*) de diferentes formas e utilizar ferramentas adicionais para fazer a desserialização do documento (passar de elementos XML para objetos). Os documentos podem ser analisados em *char*[2] ou em *byte*[3] (binário), tendo impacto no tempo de leitura e na memória utilizada.

Devido à característica desta ferramenta, que relaciona o ECU *Extract* com o esquemático, é necessário passar o esquemático para a memória antes de invocar o método *Unmarshaller*. O excerto de código seguinte representa a incorporação do esquemático do ECU *Extract* na aplicação.

```
SchemaFactory sf =
    SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
Schema schema = sf.newSchema(new File("AUTOSAR_4-3-0.xsd"));
JAXBContext jaxbContext =
    JAXBContext.newInstance("AUTOSAR_4_3_alterado");
Unmarshaller jaxbUnmarshaller = jaxbContext.createUnmarshaller();
jaxbUnmarshaller.setSchema(schema);
```

Neste caso, o esquemático é lido como um conjunto de caracteres (tipo *File*) e, para fazer a leitura do ficheiro em *bytes*, é necessário alterar a linha seguinte:

[2]Em *Java*: *char*= 16-bit(2 *bytes*); range['u0000'(0) ,'uffff'(65 535)]
[3]8-bit; range[-128,127]

```
Schema schema = sf.newSchema(new StreamSource("AUTOSAR_4-3-0.xsd");
```

A forma como o esquemático é incorporado não afeta a fase seguinte da aplicação, onde é feita a análise e validação do ECU *Extract*.

A análise e validação do ECU *Extract* é feita pela própria ferramenta de *parsing* e pode ser feita com cinco abordagens diferentes. Estas abordagens vão ter impacto no tempo e na memória utilizada nesta fase, contudo não influenciam o retorno do método. De seguida, são introduzidas as cinco combinações possíveis para a análise e validação do ECU *Extract*:

1. **JAXB(*File*):** Esta abordagem usa apenas a função *Unmarshaller* e o *input* é do tipo *File*. Assim, a leitura é feita por caracteres (*default*). Não é usada nenhuma ferramenta adicional para ajudar na associação dos elementos com as classes;

```
JAXBElement<AUTOSAR> a= (JAXBElement<AUTOSAR>)
    jaxbUnmarshaller.unmarshal(new File(FileName));
 AUTOSAR ArXml=a.getValue();
```

2. **JAXB(*StreamSource*):** Este abordagem é similar à anterior, dando-se apenas a alteração do modo como é realizada a leitura que, neste caso, passa a ser binária. Esta alteração não causa nenhum conflito com método *Unmarshaller*, contudo é alterado o tipo de ficheiro para *StreamSource*. O objeto *Streamsource* fornece uma sugestão para a implementação do JAXB para fazer a leitura em *bytes*.

```
JAXBElement<AUTOSAR> a = (JAXBElement<AUTOSAR>)
    jaxbUnmarshaller.unmarshal(new StreamSource(FileName));
AUTOSAR ArXml=a.getValue();
```

3. ***JAXB(File) + JAXBIntrospector:*** Desta vez é utilizada a ferramenta *JAXBIntrospector* que fornece acesso a dados do XML para um objeto JAXB. O intuito desta ferramenta é apenas facilitar a troca de informação independentemente da associação (se o modelo de associação é *Java* para XML ou de XML para *Java*). Na maioria das vezes o nome do elemento XML está diretamente associado com o elemento JAXB, sendo que esta ferramenta apenas facilita essa mesma associação.

```
AUTOSAR ArXml = (AUTOSAR)
    JAXBIntrospector.getValue(jaxbUnmarshaller.unmarshal(new
    File(FileName)));
```

4. ***JAXB(StreamSource) + JAXBIntrospector*:** Nesta abordagem também é usada a ferramenta auxiliar *JAXBIntrospector*. No entanto, a leitura do documento é feita em binário (*StreamSource*).

```
AUTOSAR ArXml = (AUTOSAR)
    JAXBIntrospector.getValue(jaxbUnmarshaller.unmarshal(new
    StreamSource(FileName)));
```

5. ***JAXB-StAX*:** O JAXB também permite utilizar a ferramenta de *parsing* StAX, apresentada na Sub-Secção 3.5.2 sendo que, ao usar esta ferramenta, é necessário colocar o documento de entrada obrigatoriamente no formato *StreamReader* (implica leitura de caracteres). Este método envolve mais código e mais complexidade do que qualquer outra abordagem do JAXB. Ao utilizar este modo de leitura é assumido que a análise começa no início do documento (*START_DOCUMENT*) ou no início de um elemento (*START_ELEMENT*) e a desserialização ocorre por eventos.

```
FileReader fr = new FileReader(InputFile);
XMLInputFactory xmlif = XMLInputFactory.newInstance();
 XMLStreamReader xmler = xmlif.createXMLStreamReader(fr);
 JAXBElement<AUTOSAR> a=
     jaxbUnmarshaller.unmarshal(xmler,AUTOSAR.class);
 AUTOSAR ArXml=a.getValue();
```

5.2.1 Testes de *Parsing*

Para avaliar a performance destas cinco abordagens de *parsing* foi criado um cenário de teste. O cenário implica usar um computador com um processador *Intel Core tm i5*-5300CPU 2.30 GHz com 8 GB RAM e com Windows 10. Como *inputs* da aplicação são utilizados um ECU *Extract* real (com 193 703 linhas e 8.67 MB) e o esquemático do AUTOSAR versão 4.3 (com 89 093 linhas e 5.74 MB).

Os testes consistem na análise temporal da incorporação do esquemático e do *parsing* (análise + validação) do ECU *Extract*. De forma a garantir relevância estatística, foram realizados 300 testes para cada fase.

A Figura 5.2 mostra a distribuição dos tempos obtidos na leitura do esquemático sendo que as medidas de tendência estatística são apresentadas na Tabela 5.1). Ao analisar os valores obtidos verificamos que os melhores resultados são obtidos quando a leitura é feito em binário embora a diferença seja relativamente pequena. Estes resultados coincidem com o comportamento esperado. No entanto, era esperado que a diferença fosse maior porque em binário os ficheiros são muito menores.

Tabela 5.1: Medidas de tendência estatística dos tempos de leitura do esquemático.

Tipo	**Min.**	**1st Q**	**Mediana**	**3rd Q.**	**Máx.**	**Média**	**Desvio padrão**	**90% IC**
File(char)	12.026	12.099	12.170	12.372	17.763	12.411	0.768	0.073
StreamSource(byte)	11.993	12.078	12.144	12.327	16.957	12.307	0.556	0.053

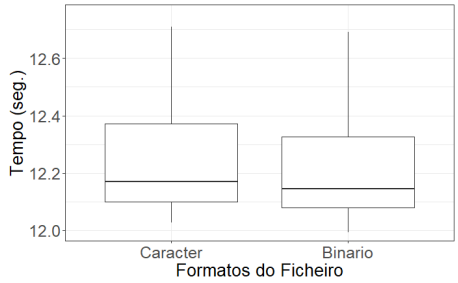


Figura 5.2: Tempos de leitura do esquemático.

Após fazer a análise dos tempos necessários para ler e incorporar o esquemático na aplicação, foram analisados os tempos para o processo de leitura do ECU *Extract* (Figura 5.3 e Tabela 5.2). Neste caso, pode ser verificado que 75% dos valores na leitura do esquemático estão entre os 11.9 segundos e os 12.3 segundos. Outro aspeto a notar é que não existe grande diferença nos tempos entre as duas abordagem, a diferença que existe são de milissegundos o que para o utilizador não é muito percetível.

Segundo os dados da Tabela 5.2 é possível verificar que o método com melhores resultados é o JAXB(*StreamSource*)+*Introspector*. Este resultado coincide com a expetativa porque era esperado que a leitura em binário fosse mais rápida e a associação entre os elementos XML e os objetos demorasse menos tempo ao utilizar o método *Iintrospector*. No entanto, para este ECU *Extract*, a diferença entre os métodos não é significativa.

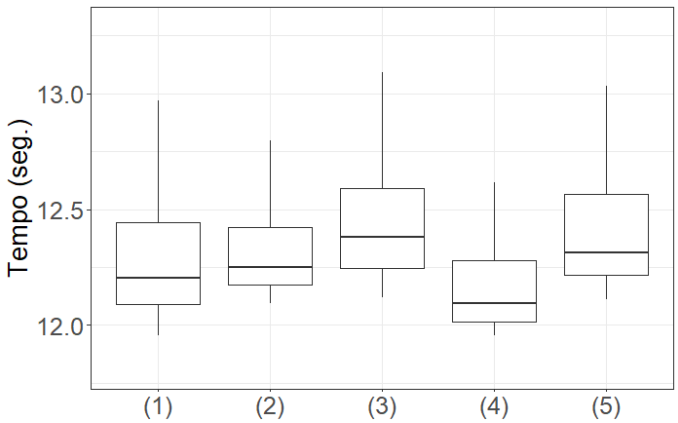


Figura 5.3: Tempos de parsing do ECU *Extract*.

Tabela 5.2: Medidas de tendência estatística dos tempos de *parsing*.

Método	**Min.**	**1st Q**	**Mediana**	**3rd Q.**	**Máx.**	**Média**	**Desvio padrão**	**90% IC**
(1) JAXB(*File*)	11.953	12.089	12.203	12.442	20.46	12.517	0.982	0.093
(2) JAXB(*StreamSource*)	12.093	12.172	12.250	12.423	16.938	12.381	0.464	0.044
(3) JAXB(*File*)+*Introspector*	12.119	12.247	12.381	12.592	18.061	12.595	0.812	0.077
(4) JAXB(*StreamSource*)+*Introspector*	11.953	12.016	12.094	12.281	16.797	12.219	0.460	0.044
(5) JAXB-*StAX*	12.109	12.215	12.312	12.566	15.640	12.552	0.625	0.059

O quando se utiliza a ferramenta auxiliar *Introspector* verificamos que os tempos de *parsing*

são menores. Este facto deve-se porque esta ferramenta permite estabelecer a ligação entre o elemento XML e o objeto mais rápido porque fornece o nome da classe associada ao elemento XML.

Segundo os tempos obtidos nas análises temporais nas diferentes etapas, foi feita a combinação dos melhores cenários e foram realizados novamente os testes para analisar os tempos dos dois passos em conjunto. Assim, a leitura dos documentos (esquemático mais ECU *Extract*) são feitas em binário e é utilizada a ferramenta *Introspector* do JAXB no *parsing*.

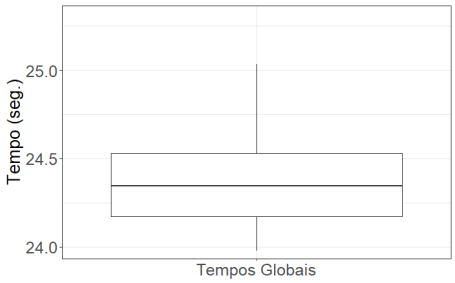


Figura 5.4: Tempos globais.

Como resultado dos testes, (Figura 5.4) temos um tempo médio de 24.526 segundos e um desvio padrão de 0.703 segundos para a validação e análise do ECU *Extract* (Tabela 5.3).

Tabela 5.3: Análise Global.

Característica	Tempo(seg.)
Min.	23.978
1st Q	24.172
Mediana	24.347
3rd Q.	24.531
Máx.	29.286
Média	24.526
Desvio padrão	0.703
Intervalo de confiança	0.067

Após fazer a análise temporal dos métodos também foi realizada a análise dos recursos utilizados pela aplicação, mais especificamente, a memória RAM. O cenário para esta análise foi o mesmo da análise temporal, também com o mesmo ECU *Extract*, o mesmo esquemático e no mesmo comutador.

A partir da análise dos recursos utilizados pela aplicação é possível verificar que a RAM utilizada no *parsing* do ECU *Extract* é praticamente igual, entre a leitura feita em binário e em caracter, apesar da Figura 5.5 poder dar a entender que a leitura em binário utiliza menos memória. Com esta análise é possível concluir que a aplicação exige menos recursos para fazer o *parsing* do que para gerar os requisitos e os casos de teste. Esta segunda parte é a que visivelmente exige mais memória é a criação dos requisitos e dos casos de testes como é possível ver na Figura 5.5. Este comportamento era o esperado porque, segundo a arquitetura da aplicação, para cada requisito gerado vai ser criado um objeto e o mesmo acontece para os casos de teste.

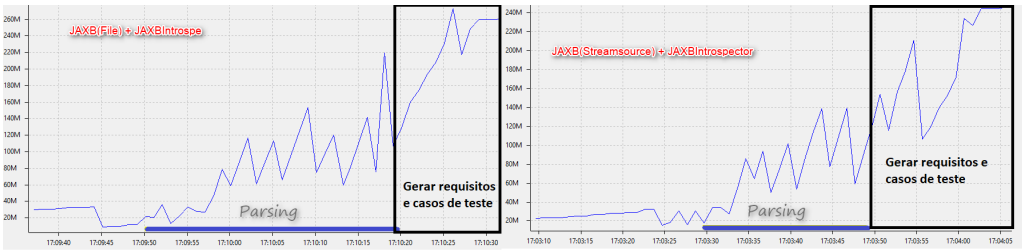


Figura 5.5: Análise da RAM.

5.2.2 Recolha de elementos e atributos

Ao fazer o *parsing* do documento, o retorno do método *Unmarshaller* é um objeto do elemento *root* que neste caso é um elemento da classe AUTOSAR. Neste objeto estão presentes todos os objetos (*Tags*) presentes no XML. Dentro de uma *Tag* é possível ter atributos, elementos (estes estão dentro da *Tag AR-PACKAGES*). Na Figura 5.6 está representado um conjunto de elementos que podem ser encontrados. Nota-se que este exemplo não tem qualquer significado real apenas tem o intuito de exemplificar a complexidade da estrutura da *framework* AUTOSAR.

A partir do esquemático é possível saber os tipos de elementos e atributos que podem ser encontrados dentro de uma *Tag*. No entanto, só em *run time* e, depois de fazer o *parsing*, é que a aplicação sabe qual é o tipo de elemento ou atributo. Um exemplo básico é o protocolo de comunicação que o ECU utiliza podendo ser *Ethernet*, CAN, *Flexray* ou vários. Só quando ocorre a desserialização do ECU *Extract* é que é determinado o tipo de objetos que foram criados. Como consequência, não é possível aceder aos atributos do objeto durante a compilação porque não há conhecimento do tipo de objeto. Inicialmente, para combater este problema foi utilizado o método de reflexão (*reflection method*). Este método permite criar e modificar o objeto de uma classe a partir de uma *string* em *run time*. Ao usar o método *reflection* o novo objeto vai herdar todos os atributos do objeto que lhe deu origem.

No excerto seguinte está presente o código do método *reflection* e o modo como é invocado o

Figura 5.6: Elementos do ECU *Extract*

método *getDATAELEMENTS*, método da classe que foi refletida. Contudo, ao implementar este método verificamos que não é o mais indicado porque implica conhecer ao detalhe todas as classes. Outra razão para não escolher esta técnica deve-se ao facto deste método ser muito vulnerável a erros porque uma alteração no nome do método de um classe resultaria em erro ao fazer a reflexão.

```
public void ReflectionMethod(File XmlFiles)
{
  Class<?> type_element;
  Object element=null;
  try {
       type_element =
           Class.forName(XmlFiles.getARPACKAGES().arpackage.get(0)
            .getELEMENTS().getACLOBJECTSETOrACLOPERATIONOrACLPERMISSION()
          .get(j).getClass().getName());
       element= type_element.newInstance();
       element=XmlFiles.getARPACKAGES().arpackage.get(0).getELEMENTS()
          .getACLOBJECTSETOrACLOPERATIONOrACLPERMISSION().get(j);

       // Call a method from the Object
       String methodName = new String("getDATAELEMENTS");
       Method getDataElementsMethod =
           element.getClass().getDeclaredMethod(methodName);
    }
}
```

Para além da dificuldade referida anteriormente, outra dificuldade encontrada foi fazer a análise dos *AR-PACKAGES* devido à complexidade e liberdade que a *framework* AUTOSAR fornece. O problema está em obter os elementos dentro da *Tag* e verificar se não existe a *Tag AR-PACKAGES* no mesmo nível hierárquico. Existem situações que o *AR-PACKAGE* não tem qualquer elemento, apenas tem o atributo com o seu nome (*SHORT-NAME*). Na tentativa de solucionar este problema decidimos aplicar um método recursivo. Este método verifica se existem *AR-PACKAGES* dentro da *Tag*. No caso de existir, a aplicação vai guardar em primeiro lugar os elementos e depois entra na *Tag AR-PACKAGES*, invocando o mesmo método. Assim, é garantido que todos os elementos são lidos.

A alternativa a esta solução era usar um método iterativo que iria percorrer o documento para encontrar os elementos. No entanto, não é possível garantir que se percorria todos os níveis dentro do *Tag* porque a profundidade (quantos níveis/camadas têm o *AR-PACKAGE*) não é conhecida. Com este método podiam surgir situações em que os requisitos não eram gerados porque haver falta falta de dados, mas esses dados estavam presentes no ECU *Extract*.

Na recolha dos elementos foi decidido que nem todos são necessários para gerar os requisitos ou relevantes para a equipa de desenvolvimento. Assim, foi decidido fazer uma filtragem, estabelecendo uma lista dos elementos de maior relevância. A implementação desta metodologia permitiu reduzir o número de elementos a analisar. Segundo a versão 4.3 do AUTOSAR existem certa de

1200 *Tags* (classes) diferentes mas, com esta filtragem analisámos apenas certa de 50 classes. A filtragem ocorre no método recursivo, verificando se os elementos da *Tag* fazem parte da lista e só depois é que entra na *Tag AR-PACKAGES*, no caso de existir. Com este método recursivo foi possível cumprir os requisitos RF6 e RF8 da Tabela 3.1)

5.3 Gerador automático de requisitos

Após termos todos os elementos relevantes do ECU *Extract* vão ser derivados os requisitos. Para isso foi necessário estabelecer padrões para os requisitos, ou seja, criar *templates* que relacionam os atributos com um tipo de requisitos. Como o processo de gerar requisitos era feito manualmente esses *templates* não estavam bem definidos havendo diferenças na escrita do mesmo requisito. Para tornar este processo *standard* foi definido que no *template* estão presentes as *Tags*, ou caminho até ao atributo. Para criar os requisitos estas *Tags* vão ser substituídas pelos respetivos valores.

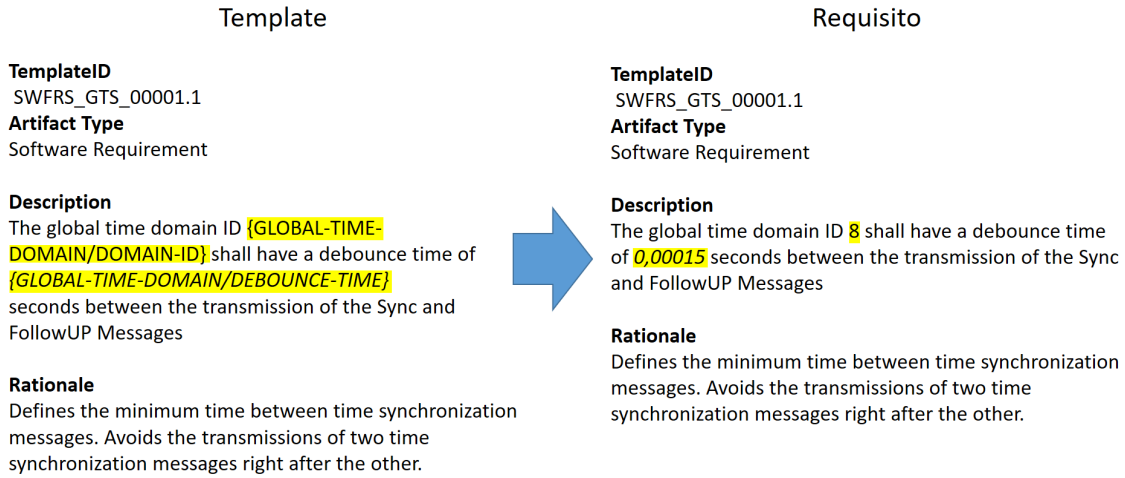


Figura 5.7: Passagem do *template* para requisito

Na Figura 5.7 está representado um exemplo da passagem de um *template* para um requisito. O requisito gerado está relacionado com o *Global Time Sync*, mais em específico com o protocolo PTP. Este *template* necessita de dois atributos: i) o identificador do domínio da rede que se encontra na *Tag GLOBAL-TIME-DOMAIN/DOMAIN-ID* no ECU *Extract* e ii) do valor correspondente ao *debounce time* (*GLOBAL-TIME-DOMAIN/DEBOUNCE-TIME*). A aplicação substitui automaticamente as *Tags* pelos respetivos valores, ou seja, no caso *GLOBAL-TIME-DOMAIN/DOMAIN-ID* por oito e no *GLOBAL-TIME-DOMAIN/DEBOUNCE-TIME* por 0,00015.

Para validar este processo foi necessário inicialmente validar os *templates*, verificar se cumprem as regras impostas pelo RE, se não havia nenhum erro de escrita e se as *Tags* estavam corretas. Após a verificação dos *templates* foi verificado se o requisitos estavam associados aos elementos corretos e se todos os campos foram substituídos corretamente.

Na aplicação, até ao momento da escrita desta dissertação, foram implementados 29 *templates* o que originou 1259 requisitos para o ECU *Extract* utilizado no cenário de testes referido anteriormente.

	Primary Test	Artifact Type	State	state	Source File	Verification Type
2	The POS shall provide a Controller Area Network as a communication interface.	System Requirement	New	Pending	POS_STAR_3_2020_Ecu_Extract_2018_05a_A R43.arxml	Software Test
3	The CAN interface shall have the following configurations: CAN Baudrate: 500000; Protocol Name: CAN; Protocol Version: can2.0b; CAN-FD Baudrate: 1000000; Wake-UP Support: true; Maximum Time Quanta Per Bit: 40; Maximum Sample Point: 8000; Maximum Sync Jump Width: 2000; Minimum Number of Time QuantaPer Bit: 40; Minimum Sample Point: 8000; Minimum Sync Jump Width: 20.0;	Software Requirement	New	Pending	POS_STAR_3_2020_Ecu_Extract_2018_05a_A R43.arxml	Software Test

Figura 5.8: Lista de requisitos

Na Figura 5.8 está presente um excerto do ficheiro gerado automaticamente pela aplicação com todos requisitos (requisitos RF9 e RF10 da Tabela 3.1). Cada coluna representa características do requisito, tais como, o tipo de requisito, o estado, o documento de onde foi derivado (nome do ECU *Extract*), o estado da revisão do requisito e o tipo de verificação que é necessário fazer (estrutura definida no RNF1 da Tabela 3.1). Com a lista de requisitos gerados é possível gerar os casos de teste e caso não existam requisitos não é possível ter casos de teste. Por questões de confidencialidade foram omitidos alguns características do ECU que foi analisado. No entanto, neste excerto é possível ver algumas característica do protocolo CAN utilizado pelo ECU, cumprindo os requisitos RNF2 e RF11 o da Tabela 3.1.

5.4 Gerador automático de descrições de casos de teste

Para a criação dos casos de teste foram criados *templates*, seguindo a mesma metodologia do processo para a geração de requisitos. Para cada tipo de requisito (para cada *template* de requisito) está associado um tipo de caso de teste (*Template* de caso de teste). Na Figura 5.9 são apresentadas as relações entre os tipos de casos de teste e os tipos de requisitos. A relação, neste caso, é de um caso de teste para dois tipos de requisitos mas, é possível ter relações de 1:1 ou 1:N. A ligação entre os tipos de teste e os requisitos é feita através dos identificadores dos *templates*.

Até este momento de escrita da dissertação, foram criados sete *templates* para os casos de teste relacionados com os requisitos do *Global Time Sync*. Estes tipos de *templates* podem gerar casos de testes para o protocolo CAN ou para *Ethernet*.

Como é possível ver na Figura 5.9, o teste com a identificação *TS_GTS_0001* está relacionado com dois tipos de requisitos, i) o *template* com a identificação *SWFRS_GTS_0001.12* (requisito de *software*-SWFRS- sobre *Golbal Time Sync*-GTS- em CAN) e ii) o requisito *SWFRS_GTS_0002.12* (requisito de *software*- SWFRS - sobre *Golbal Time Sync*-GTS- em *Ethernet*).

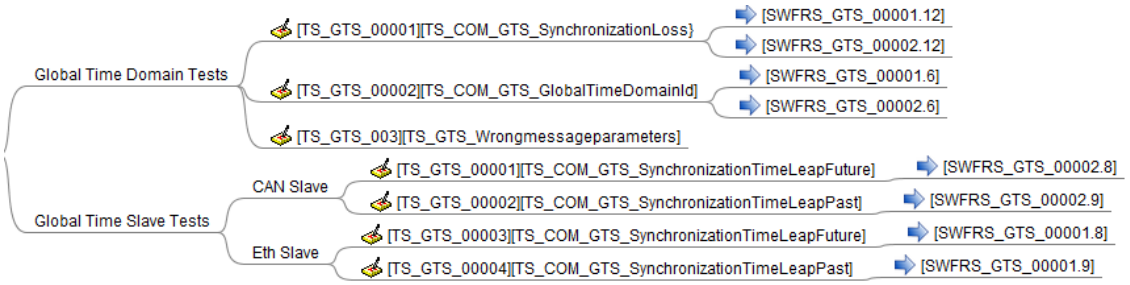


Figura 5.9: Relação entre *templates* de teste e *templates* de requisitos

Um teste realizado pela aplicação para validar o processo é se nos casos de testes gerados é encontrado o caracter "{". Caso seja encontrado, significa que existe uma *Tag* que não foi devidamente substituída sendo eliminado da lista de casos de teste gerados. Para validar a aplicação também foi verificado se a substituição das *Tags* está correta e se os requisitos relacionados são os corretos, ou seja, o *template* do qual o requisito derivou está presente na lista de requisitos relacionados com o teste.

Test Specification Name	Desciprion	Precondiction	Post-Condiction	Step List
TS_COM_GTS_SynchronizationLoss	During time synchronization procedure a synchronization loss can happen between a Virtual Local Time Base and a Global Time Base.	1 - The test procedure can control when the Global Time Master is sending the time synchronization messages. 2 - The Global Time Slave is able to receive the Time Synchronization messages and synchronize it's Virtual Local Time Base with the Global Time Base.	1 - Make sure that the Global Time Master is enabled again at the end of the test procedure	**Step** 1: Description: Enable the transmission of the time synchronization messages Expect Results: Pass-Criteria: The time synchronization messages are not being transmitted Fail-Criteria: The time synchronization messages are transmitted from the global time master to the global time slave **Step 2** : Description: Stop the transmission of the time synchronization messages Expect Results: Pass-Criteria: If the time synchronization messages are being transmitted Fail-Criteria: If the time synchronization messages are not being transmitted any mo **Step 3:** Description: Wait for 1.75 seconds, and use the diagnostic service ReadDataByIdentifier (0x22) to request the data (0x0107). Expect Results: Pass-Criteria: If the diagnostic service ReadDataByIdentifier returns a negative response Fail-Criteria: If the diagnostic service ReadDataByIdentifier returns a

Figura 5.10: Especificação de um caso de teste

Na figura 5.10 está presente um excerto do ficheiro criado pela aplicação com a descrição dos testes relacionadas com o *Global Time Sync*, em concreto com o PTP (por uma questão de confidencialidade foram omitido os tempos nos casos de teste). É apresentado o nome do teste, uma pequena descrição das condições necessárias, o objetivo do teste e a sequência de passos com as respetivas condições de validação.

5.5 Resumo do Capítulo

Neste capítulo foram apresentadas as diferentes funções do gerador automático de requisitos e descrições de casos de teste. Inicialmente explicamos o modo como é feita a análise do ECU *Extract*, realizando testes realizados para fazer uma avaliação da performance da aplicação. De seguida,mostrámos como se fez a ligação entre os elementos do ECU *Extract* e os requisitos e a ligação entre os requisitos e os casos de teste.

Por fim, explicamos como se passa de um *template* para um requisito, tendo apresentado excertos dos requisitos gerados a partir de 29 tipos (*templates*). O mesmo é feito para os casos de teste tendo explicado os processos de validação dos mesmos.

Capítulo 6

Conclusões e Trabalhos Futuros

6.1 Conclusões

Cada sistema de controlo automóvel, baseado num ECU, é descrito por uma lista de especificações lógicas (requisitos) que traduz as especificações técnicas do ECU descritas no documento ARXML para um formato mais intuitivo. Os sistemas de controlo são cada vez mais complexos e estão sempre a sofrer *updates*. Isto torna a criação da lista de requisitos que cada ECU tem de cumprir um processo cada vez mais difícil, demorado e com elevada probabilidade de erros humanos.

Na tentativa de combater este problema, foi proposta nesta dissertação uma aplicação para tornar este processo automático de forma a que a passagem das especificações técnicas presentes no ECU *Extract* para requisitos não levante dúvidas sobre as características e funcionalidades do sistema de controlo.

Como esta dissertação foi desenvolvida em ambiente empresarial, nomeadamente na empresa da BOSCH®, a solução foi direcionada para as necessidades desta empresa. Assim sendo, inicialmente foi necessária a integração com a equipa de desenvolvimento bem como estudar e compreender a estrutura da *framework* AUTOSAR, que é a *framework* usada pela BOSCH® e muitas outras empresas na descrição de sistemas de controlo automotivos.

Após o levantamento de ferramentas a serem utilizadas e de um cuidado planeamento, foi desenvolvida uma aplicação que gera automaticamente requisitos e descrições de casos de teste referentes a um ECU *Extract*. Assim sendo, a aplicação analisa um ECU *Extract* e extrai as informações mais relevantes do sistema de controlo, tais como, características dos protocolos de comunicação, características das mensagens trocadas, serviços suportados, etc.. Depois de obter estas informações a aplicação elabora a lista de requisitos e descrições de casos de teste para a validação dos mesmos. Nos casos de teste estão incluídos os cenários e as condições para se testar um determinado requisito extraído.

Durante a elaboração da aplicação, as maiores dificuldades prenderam-se com o *parsing* do ECU *Extract* e a geração dos requisitos. Relativamente ao *parsing* a dificuldade deveu-se ao facto do ECU *Extract* não seguir a estrutura habitual de um documento XML. Outra dificuldade está

relacionada com a seleção da informação que vai ser recolhida do ECU *Extract*, pois nem toda informação é necessária. Desta forma, foi necessário estabelecer métodos de pesquisa específicos para o problema. No que diz respeito à geração de requisitos, foi necessário ultrapassar a inexistência de *templates* que definiam o tipo de requisito. Desta forma, foi necessário estabelecer diferentes tipos de *templates* que definem a estrutura e a relação entre requisitos e as características do sistema.

O desenvolvimento desta dissertação numa empresa com a dimensão da BOSCH® levantou alguns desafios. Um deles foi o processo de integração com a equipa de desenvolvimento devido ao facto de ser a primeira vez que a equipa integrava um elemento a realizar uma dissertação de mestrado. Contudo, este desafio foi ultrapassado com sucesso, principalmente com a mudança para o novo edifício onde o meu local de trabalho estava mais próximo do resto da equipa. O outro grande desafio deveu-se às inúmeras formações, processos internos de planeamento e reuniões que acabavam por vezes por retirar algum tempo ao desenvolvimento da aplicação. No entanto, estes processos ajudaram a desenvolver competências relacionados com a gestão de tempo, organização de projetos e de recursos humanos. Além disso, a BOSCH® disponibilizou também formações relacionados com a segurança no trabalho, os processos de produção implementados na empresa e formações internas relacionadas com tópicos específicos. Um desses tópicos foi a *framework* AUTOSAR pois devido à sua elevada complexidade a empresa achou necessário criar uma formação com duração de três dias com o intuito de aumentar o conhecimento dos seus colaboradores e esclarecer dúvidas sobre essa *framework*.

No final, a aplicação desenvolvida nesta dissertação cumpre todos os objetivos inicialmente propostos. Além disso, o fato de ter sido desenvolvida num ambiente empresarial e numa equipa de desenvolvimento bastante dinâmica trouxe uma grande evolução às minhas competências pessoais e profissionais.

6.2 Trabalhos futuros

Nesta última secção identificam-se alguns aspetos que podem ser desenvolvidos de forma a melhorar a aplicação. Um desses aspetos é a relação entre a informação recolhida do ECU *Extract* e a lista de requisitos gerados que pode ainda ser melhorada através da implementação de mais *templates*. Neste caso, o uso de uma interface gráfica poderá facilitar a interação com o utilizador. O mesmo acontece para os casos de teste.

A aplicação foi testada com dois ECUs *Extract* de diferentes clientes. Contudo, é necessário validar a aplicação com mais ECUs *Extract* para tornar a aplicação mais robusta.

Por último, um dos requisitos da aplicação era providenciar os *outputs* em formato *Excel*, o qual foi implementado com sucesso. Contudo, visto que existe uma base de dados que faz a gestão dos requisitos relacionados com ECUs *Extract* já implementada na BOSCH®, seria vantajoso que os *outputs* da aplicação integrassem diretamente este sistema de gestão de requisitos.

Capítulo 7

Anexos

7.1 Método recursivo

```
public static void RecursiveArPackage(ARPACKAGE ArP) throws
    ClassNotFoundException, InstantiationException,
    IllegalAccessException, NoSuchMethodException, SecurityException,
    IllegalArgumentException, InvocationTargetException {

      if (logger.isDebugEnabled()) {
      logger.debug("entering RecursiveArPackage(ARPACKAGE)");
      logger.debug("ArP: " + ArP);
    }
      List<Object> ARObjects = new ArrayList<Object>();

      if(ArP.getARPACKAGES()!=null) {
        List<ARPACKAGE> ARPackages = new ArrayList<ARPACKAGE>();
        // get the elements from the AR-Package
        if(ArP.getELEMENTS()!=null) {
          ARObjects
              =ArP.getELEMENTS().getACLOBJECTSETOrACLOPERATIONOrACLPERMISSION();
          for (int j=0;j<
              ArP.getELEMENTS().getACLOBJECTSETOrACLOPERATIONOrACLPERMISSION().size();j
              {

            addElementToList(ARObjects.get(j));
          }
        }
        ARPackages=ArP.getARPACKAGES().getARPACKAGE();

        //exist others AR-PACKAGE inside the AR-PACKAGE
        for (ARPACKAGE arpackage : ARPackages) {
          RecursiveArPackage(arpackage);
```

```
        }
      }
      else {
        // get the elements from the last AR-Package
        ARObjects
            =ArP.getELEMENTS().getACLOBJECTSETOrACLOPERATIONOrACLPERMISSION();
        for (int j=0;j<
            ArP.getELEMENTS().getACLOBJECTSETOrACLOPERATIONOrACLPERMISSION().size();j++)
            {

          addElementToList(ARObjects.get(j));
        }
        if (logger.isDebugEnabled()) {
          logger.debug("exiting RecursiveArPackage()");
        }
        return;
      }
  }
```

Referências

S Balaji. Waterfall vs v-model vs agile : A comparative study on SDLC. *WATEERFALL Vs V-MODEL Vs AGILE : A COMPARATIVE STUDY ON SDLC*, 2(1):26–30, 2012. ISSN 2304-0777. doi: 10.1.1.695.9278.

Lucia Lo Bello. The case for ethernet in automotive communications. *ACM SIGBED Review*, 8 (4):7–15, 2011. ISSN 15513688. doi: 10.1145/2095256.2095257.

R Bruckmeier. Ethernet for automotive Applications. *Fresscale technology Forum*, 2010.

Ece Calikus. Mobile App Analytics & Sentiment Analysis of Master of Science in Computer Science Mobile App Analytics & Sentiment Analysis of Customer. (October 2015), 2016. doi: 10.13140/RG.2.1.2072.3922.

Ernst Christmann. Data Communication in the Automobile – Part 2 :. páginas 2–6, 1993. URL `https://elearning.vector.com/portal/medien/cmc/press/PTR/SerialBusSystems{_}Part2{_}ElektronikAutomotive{_}200612{_}PressArticle{_}EN.pdf`.

Peter Decker. CAN Gets Even Better. (April):1–4, 2013. URL `https://elearning.vector.com/portal/medien/cmc/press/Vector/CAN{_}FD{_}ElektronikAutomotive{_}201304{_}PressArticle{_}EN.pdf`.

Joseph Fialli. The Java™ Architecture for XML Binding (JAXB) Java™ Architecture for XML Binding (JAXB) Specification ("Specification "). 2003. URL `http://download.oracle.com/otn-pub/jcp/7196-jaxb-1.0-fr-spec-oth-JSpec/jaxb-1{_}0-fr-spec.pdf?AuthParam=1519388982{_}55f58ae684e7f3feb3a2360efd90dd86`.

Marcus Hammarberg e Joakim Sunden. *Kanban in action*. Manning Publications Co., 2014. ISBN 1617291056.

Su Cheng Haw e G S V Radha Krishna Rao. A comparative study and benchmarking on xml parsers. In *Advanced Communication Technology, The 9th International Conference on*, volume 1, páginas 321–325. IEEE, 2007. ISBN 8955191316.

Harald Heinecke, Klaus-Peter Schnelle, Helmut Fennel, Jürgen Bortolazzi, Lennart Lundh, Jean Leflour, Jean-Luc Maté, Kenji Nishikawa e Thomas Scharnhorst. Automotive open system architecture-an industry-wide initiative to manage the complexity of emerging automotive e/e-architectures. Relatório técnico, 2004.

Ris Hvanth, D. Valli e K. Ganesan. Design of an In-Vehicle Network (Using LIN, CAN and FlexRay), Gateway and its Diagnostics Using Vector CANoe. *American Journal of Signal*

Processing, 1(2):40–45, feb 2012. ISSN 2165-9354. doi: 10.5923/j.ajsp.20110102.07. URL http://article.sapub.org/10.5923.j.ajsp.20110102.07.html.

Andreas Kern, Hongyan Zhang, Thilo Streichert e Jurgen Teich. Testing switched Ethernet networks in automotive embedded systems. In *2011 6th IEEE International Symposium on Industrial and Embedded Systems*, páginas 150–155. IEEE, jun 2011. ISBN 978-1-61284-818-1. doi: 10.1109/SIES.2011.5953657. URL http://ieeexplore.ieee.org/document/5953657/.

U. Kiencke. A view of automotive control systems. *IEEE Control Systems Magazine*, 8(4):11–19, 1988. ISSN 0272-1708. doi: 10.1109/37.7725. URL http://ieeexplore.ieee.org/document/7725/.

Uwe Kiencke e Lars Nielsen. *Automotive control systems: For engine, driveline, and vehicle: Second edition*. 2005. ISBN 3540231390. doi: 10.1007/b137654.

Tak Lam Tak Lam, J.J. Ding e Jyh-Charn Liu Jyh-Charn Liu. XML Document Parsing: Operational and Performance Characteristics. *Computer*, 41(9):30–37, 2008. ISSN 0018-9162. doi: 10.1109/MC.2008.403.

J. Lang. ISO 13400 - Diagnostics over Internet Protocol (DoIP), 2010.

Robert C Martin. *Agile software development: principles, patterns, and practices*, volume null. Prentice Hall, 2002. ISBN 0135974445.

Eugen Mayer. Serial Bus Systems in the Automobile. páginas 1–5, 2006a. URL https://elearning.vector.com/portal/medien/cmc/press/PTR/SerialBusSystems{_}Part4{_}ElektronikAutomotive{_}200703{_}PressArticle{_}EN.pdf.

Eugen Mayer. Serial Bus Systems in the Automobile. *Vector Informatik GmbH*, páginas 1–6, 2006b. URL https://elearning.vector.com/portal/medien/cmc/press/PTR/SerialBusSystems{_}Part2{_}ElektronikAutomotive{_}200612{_}PressArticle{_}EN.pdf.

Nicolas Navet, Ye-qiong Song, Françoise Simonot-lion e Cédric Wilwert. Trends in Automotive Communication Systems. página 21, 2007.

Thomas Nolte, Hans Hansson e Lucia Lo Bello. Automotive Communications - Past, Current and Future. 1:985–992, 2005. URL https://ieeexplore.ieee.org/stamp/stamp.jsp?tp={&}arnumber=1612631.

Thorsten Piper, Stefan Winter, Paul Manns e Neeraj Suri. Instrumenting AUTOSAR for dependability assessment: A guidance framework. *Proceedings of the International Conference on Dependable Systems and Networks*, 2012. ISSN 1530-0889. doi: 10.1109/DSN.2012.6263913.

B. Regnell, K. Kimbler e A. Wesslen. Improving the use case driven approach to requirements engineering. In *Requirements Engineering, 1995., Proceedings of the Second IEEE International Symposium on*, páginas 40–47, Mar 1995. doi: 10.1109/ISRE.1995.512544.

Ralph Santitoro. Metro Ethernet Services – A Technical Overview What is an Ethernet Service. *Metro*, páginas 1–19, 2003. URL http://metroethernetforum.org/PDF{_}Documents/metro-ethernet-services.pdf.

Ken Schwaber. *Agile project management with Scrum*. Microsoft press, 2004. ISBN 0735637903.

K. Senthilkumar e Ramesh Ramadoss. Designing multicore ECU architecture in vehicle networks using AUTOSAR. In *3rd International Conference on Advanced Computing, ICoAC 2011*, páginas 270–275, 2011. ISBN 9781467306690. doi: 10.1109/ICoAC.2011.6165187.

ISO Standard. Iso 11898, 1993. *Road vehicles–interchange of digital information–Controller Area Network (CAN) for high-speed communication*, 1993.

Mayer Eugen © 2010-2018 Vector Informatik GmbH. Introduction to Automotive Safety, 2014. URL `https://elearning.vector.com/vl{_}automotive{_}ethernet{_}introduction{_}en.html`.

Fang Zhou, Shuqin Li e Xia Hou. Development method of simulation and test system for vehicle body CAN bus based on CANoe. In *Intelligent Control and Automation, 2008. WCICA 2008. 7th World Congress on*, páginas 7515–7519. IEEE, 2008. ISBN 1424421136.