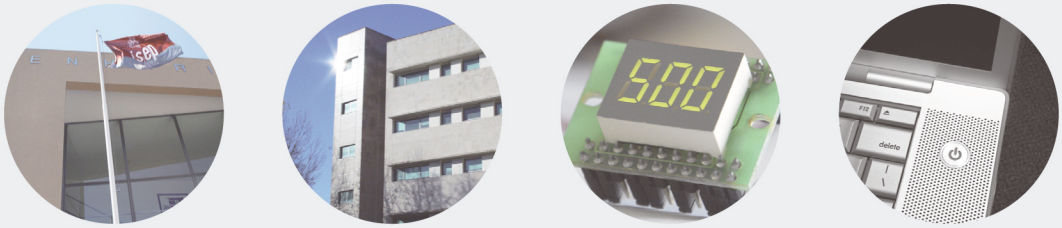
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES



Desenvolvimento de um Parser de ficheiros ARXML

NUNO MIGUEL SILVA PACHECO
dezembro de 2021

POLITÉCNICO
DO PORTO

Desenvolvimento de um Parser de ficheiros ARXML

Nuno Miguel Silva Pacheco

Mestrado em Engenharia Eletrotécnica e de Computadores
Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto
2021

Este relatório satisfaz, parcialmente, os requisitos que constam da Ficha de Unidade Curricular de Tese/Dissertação, do 2º ano, do Mestrado em Engenharia Eletrotécnica e de Computadores

Candidato: Nuno Pacheco, Nº 1161160, 1161160@isep.ipp.pt
Orientação científica: Paula Viana, pmv@isep.ipp.pt
Empresa: Continental Engineering Services
Supervisão: Carlos Garcia, carlos.daniel.garcia@conti-engineering.com

Mestrado em Engenharia Eletrotécnica e de Computadores
Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto
10/11/2021

Agradecimentos

Os meus agradecimentos vão para todos aqueles que, de diferentes modos, apoiaram-me na realização desta dissertação. Gostaria, no entanto, de destacar aqueles que mais diretamente colaboraram comigo. Em especial, ao Engenheiro Carlos Garcia pela oportunidade de realização da dissertação na Continental Engineering Services, disponibilidade e encorajamento que me demonstrou na sua realização. À Engenheira Emilie Le Biller pela sua disponibilidade, encorajamento e ajuda. À Professora Paula Viana por ter aceitado ser minha orientadora e pela sua orientação. A todas as pessoas (professores/colegas) que partilharam o seu conhecimento ao longo destes anos que de alguma forma me ajudaram a chegar até aqui, contribuído de forma essencial para a minha formação, tanto a nível académico, como a nível humano. Aos meus pais, pela paciência, apoio incondicional e a força que me deram para realizar esta dissertação. E, por fim, aos meus amigos mais próximos por me terem acompanhado nos bons e nos maus momentos.

Resumo

Uma das maiores preocupações nas grandes empresas é a mitigação dos custos. Esta restrição está muito presente na indústria automóvel devido ao uso de *third party softwares*, tornando-se assim essencial o desenvolvimento de estratégias sem a aquisição de licenças de ferramentas necessárias para o desenvolvimento de *software*. A monitorização do tráfego da comunicação entre os diferentes componentes num sistema automóvel é muitas vezes essencial para os *developers* testarem o *software* desenvolvido, sendo isto, geralmente, feito com o auxílio de *third party softwares*, o que, inevitavelmente, requer um número significativo de licenças. Sendo assim, foi desenvolvido um programa em C++, com a utilização de uma biblioteca license free, para parsing de bases de dados do tipo ARXML (AUTOSAR XML) com o principal objetivo de se registar, entender e visualizar a informação lá contida. Para isso é feita a integração numa framework desenvolvida pela Continental Engineering Services de modo a esta se comportar como a Graphical User Interface (GUI).

Palavras-Chave: AUTOSAR, Parser, C++, Framework, ARXML, License Free

Abstract

One of the biggest concerns in large companies is cost mitigation. This restriction is very present in the automotive industry due to the use of third party software, thus making it essential to develop strategies without acquiring licenses of ownership of tools that meet the specific needs during the development of software. Monitoring the communication traffic between different components in an automotive system is often essential for developers to test the projected software, and this is usually done with the help of third party software, which inevitably requires a significant number of licenses. Therefore, a script in C ++ was developed, using a free license library, for parsing ARXML-type databases (AUTOSAR XML) with the main objective of registering, understanding and visualizing the information contained in them. For that, the script is integrated in a framework developed by Continental Engineering Services so that it behaves like a GUI.

Keywords: AUTOSAR, Parser, C++, Framework, ARXML, License Free

Índice

Índice de Figuras

Acrónimos

AA Adaptive Applications

ADAS Advanced Driver-Assistance Systems

ARA AUTOSAR Runtime for Adaptive Applications

AUTOSAR AUTomotive Open System ARchitecture

BMW Bayerische Motoren Werke

BSW Basic Software

CAN Controller area network

CDD Complex Device Drivers

CES Continental Engineering Services

CRC Cyclic Redundancy Code

DCM Diagnostic Communications Manager

DEE Departamento de Engenharia Eletrotécnica

DEM Diagnostic Event Manager

DoIP Diagnostics over Internet Protocol

ECU Electronic control unit

E/E Eletrical/Eletronic

GUI Graphical User Interface

HPC High Performance Computing

IoT Internet of Things

ISEP Instituto Superior de Engenharia do Porto

LAN Local Area Network

MCAL Microcontroller Abstraction Layer

MEEC Mestrado em Engenharia Eletrotécnica e de Computadores

OEM Original Equipment Manufacturers

PDU Protocol Data Unit

RTE Runtime Environment

SCC Smart Charge Communication

SOA Service Oriented Architecture

SoAd Socket Adaptor Module

TCP Transmission Control Protocol

TEDI Tese/Dissertação

UDP User Datagram Protocol

VFB Virtual Functional Bus

WAN Wide Area Network

WLAN Wireless Local Area Network

Capítulo 1

Introdução

1.1 Contextualização

Este trabalho surge no âmbito de Tese/Dissertação (TEDI), unidade curricular do segundo ano do Mestrado em Engenharia Eletrotécnica e de Computadores (MEEC) do Departamento de Engenharia Eletrotécnica (DEE), que pretende proporcionar aos estudantes a aplicação integradora dos conhecimentos adquiridos ao longo do curso e da proposta de estágio da Continental Engineering Services (CES).

A CES foi fundada a 18 de Novembro de 1997[22], sendo a sua empresa mãe a Continental AG, uma empresa internacional alemã de produção de peças no setor automóvel, especializada em sistemas de travagem, eletrónica de interiores, segurança automóvel, componentes de motor e chasis, tacógrafos e pneus, entre outras. A sua sede está localizada em Hannover e é o quarto maior fabricante de pneus do mundo [4]. A CES tornou-se internacional e tem-se vindo a expandir desde 2006, estando agora presente em 23 localizações diferentes na Europa, América do Norte, China e Japão, com mais de 1800 colaboradores.

A CES tem como missão e visão, como uma empresa centrada em tecnologia, encontrar soluções adequadas para todos os desafios, atuando pragmaticamente e reagindo rapidamente a todas as inovações tecnológicas, influências do mercado e requisitos do cliente, sendo que a equipa de Advanced Driver-Assistance Systems (ADAS) trabalha em projetos relacionados com o auxílio ao condutor [22].

Uma das maiores preocupações nas grandes empresas é a mitigação dos custos fixos. Esta restrição está muito presente na indústria automóvel devido ao uso de *third party softwares*, tornando-se assim essencial o desenvolvimento de estratégias sem a aquisição de licenças de ferramentas necessárias para o desenvolvimento de *software*. A monitorização do tráfego da comunicação entre os

diferentes componentes num sistema automóvel é muitas vezes essencial para os *developers* testarem o *software* desenvolvido, sendo isto, geralmente, feito com o auxílio de *third party softwares*, o que, inevitavelmente, requer um número significativo de licenças. Sendo assim, surgiu a necessidade de se desenvolver um *software* para parsing de bases de dados do tipo ARXML (AUTOSAR XML) capaz de visualizar, registar e entender o tráfego de dados baseado nessa base de dados.

1.2 Objetivos

Este trabalho tem como objetivo o desenvolvimento de uma aplicação que permita fazer o *parsing* de ficheiros ARXML, de modo a se obter informação através de Controller area network (CAN) Frames, para depois ser integrada na framework desenvolvida pela empresa, tendo como suporte os seguintes pontos:

- Obtenção dos diferentes Frames;
- Obtenção dos diferentes campos essenciais dos Frames;
- Integração do script na core da framework;
- Utilização da framework como GUI.

1.3 Organização do relatório

O presente relatório apresenta-se organizado segundo uma ordem coerente dos conteúdos abordados. No segundo capítulo é feito um estudo acerca do AUTOSAR. No terceiro capítulo é feita uma investigação de qual software utilizar. No quarto capítulo é demonstrado e explicado o código desenvolvido. No quinto capítulo são demonstrados os resultado obtidos e, finalmente, no sexto é feita a conclusão do relatório.

Capítulo 2

AUTOSAR (AUTomotive Open System ARchiteture)

A arquitetura lógica dos sistemas de software automóvel é responsável pelas funcionalidades mais avançadas dos veículos, como por exemplo, a travagem automática quando um peão é detetado na trajetória do automóvel, o veículo ligar os faróis médios quando fica de noite ou escovas automáticas nos dias de chuva. Estas funcionalidades são geralmente realizadas por componentes lógicos de software, nos exemplos anteriores, os sensores detetam a dita alteração e solicitam ação automática ao componentes responsáveis pelas funcionalidades. Estes elementos comunicam e trocam informação entre si. Para além disso, os elementos são agrupados em subsistemas, com base nos seus tipos de funcionalidades. A arquitetura física destes sistemas é geralmente distribuída por vários Electronic control unit (ECU)s. Estes controladores são conectados por barramentos, por exemplo, CAN ou *Ethernet*, e são responsáveis por executar as várias funcionalidades mencionadas anteriormente. Para facilitar o desenvolvimento dos sistemas de software automóvel, introduziu-se a norma AUTOSAR em 2003 como uma parceria de vários Original Equipment Manufacturers (OEM)s automóvels. Hoje, consiste em mais de 150 parceiros globais e é considerado a norma na indústria automóvel. O AUTOSAR foi concebido com os seguintes objetivos:

- Uniformização da arquitetura de referência do ECU e das suas camadas, aumentando a capacidade de reutilização dos componentes de software usados em diferentes projetos desenvolvidos pelos mesmos fornecedores, representado na figura 2.1;

- Uniformização da metodologia de desenvolvimento de projetos, o que per-

mite a colaboração entre vários OEMs diferentes no processo de desenvolvimento de software para todas os ECUs no sistema;

- Uniformização dos modelos arquitetónicos do sistema, permitindo uma troca fácil de modelos de arquitetura entre diferentes ferramentas usadas por diferentes colaboradores no processo de desenvolvimento do projeto;
- Uniformização do middleware, arquitetura e funcionalidade dos ECUs, o permite que aos engenheiros dos OEMs se concentrarem apenas no projeto e na implementação das funcionalidades de alto nível nos veículos [24].

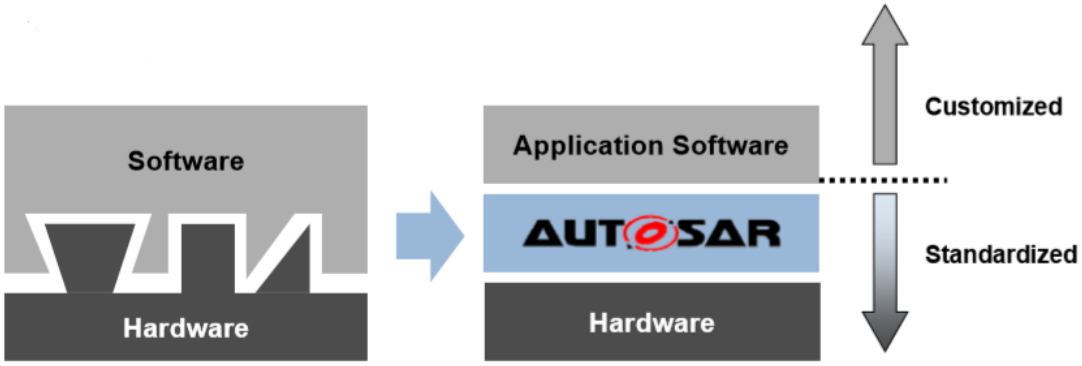


Figura 2.1: Antes e depois do AUTOSAR [7]

2.1 História

A parceria do desenvolvimento do AUTOSAR foi formada em julho de 2003 pela Bayerische Motoren Werke (BMW), *Robert Bosch*, Continental AG, *Daimler* AG (anteriormente *Daimler-Benz*), *Siemens* VDO e *Volkswagen* de modo a se desenvolver e estabelecer uma norma aberto da indústria para arquitetura automóvel. Em novembro de 2003, juntou-se também a *Ford Motor Company* e, em dezembro, entrou também o *Groupe* PSA (anteriormente PSA *Peugeot Citroën*) e a *Toyota Motor Corporation*. Em 2004, a *General Motors* também se juntou à parceria [1]. Desde 2003, desenvolveram-se quatro versões principais da *Classic Platform*. O desenvolvimento do AUTOSAR pode ser dividido em três fases:

- Fase 1 (2003–2006): Desenvolvimento inicial da norma (versões 1.0, 2.0, 2.1);

- Fase 2 (2007–2009): Melhoria da norma em termos de arquitetura e metodologia (versões 3.0, 3.1, 4.0);

- Fase 3 (2010–2012): Manutenção e melhorias específicas (versão 3.2)[21].

A partir de 2013 o consórcio entrou num modo de trabalho contínuo para a *Classic Platform*, de modo a manter a norma e fornecer as melhorias necessárias,

lançando as versões 4.1 (2013), 4.2 (2014) e 4.3(2016). Em 2016, começaram a trabalhar na *Adaptive Platform*. A primeira versão foi publicada no início de 2017, seguida de outros lançamentos em outubro do mesmo ano, março de 2018 e outubro de 2018 [1]. Devido ao desenvolvimento do coronavírus (Covid-19) e ao seu impacto nos dias de hoje, a comunidade AUTOSAR fez a sua primeira conferência virtual de lançamento da última versão do AUTOSAR a 4 de dezembro de 2020, lançando assim a versão mais recente R20-11[19].

2.2 AUTOSAR Classic Platform

Como foi mencionado, o aumento da complexidade dos veículos modernos e, especialmente, dos seus sistemas E/E foi a principal motivação por trás do desenvolvimento do AUTOSAR. Além disso, os veículos da atualidade têm mais de cem ECUs instalados. Sem a norma do AUTOSAR, estes iriam ter que ser refeitos quando o *hardware* fosse alterado. Sendo assim, foi esta a razão pela qual os gigantes da indústria automóvel se juntaram e tornaram o *software* independente do *hardware*.

Um conceito essencial desta plataforma é o Virtual Functional Bus (VFB). Este barramento virtual separa as aplicações da infraestrutura e comunica através de *ports* dedicados, o que significa que as interfaces de comunicação do *software* são mapeadas para esses *ports*, representado na figura 2.2. O VFB lida com a comunicação dentro de cada ECU individualmente e entre os diferentes ECUs. Do ponto de vista da aplicação, não é necessário conhecimento detalhado de tecnologias ou dependências de nível inferior, suportando o desenvolvimento independente de *hardware* e o uso de *software* de aplicação [2].

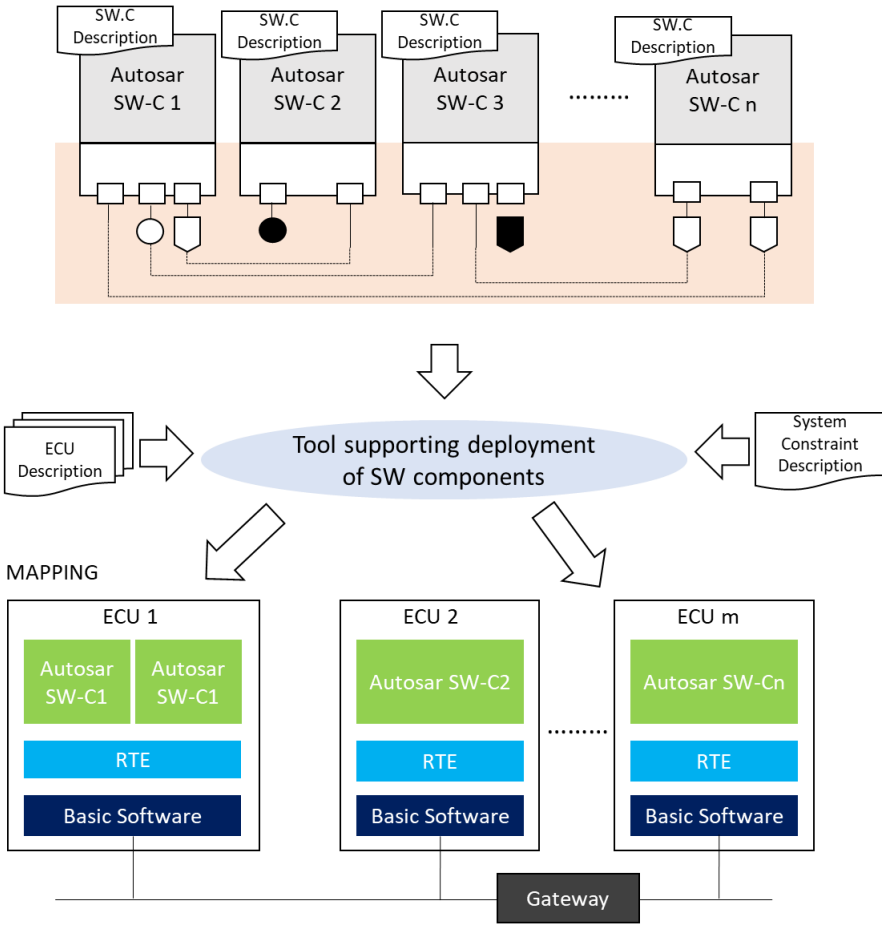


Figura 2.2: Comunicação com o Virtual Function Bus[5]

A arquitetura em camadas do AUTOSAR, representada na figura 2.3, oferece todos os mecanismos necessários para a independência do software e hardware. Está dividida em três camadas principais: *Application Layer*, Runtime Environment (RTE) e Basic Software (BSW). O seu principal conceito é a separação do *software* das aplicações para *hardware* e do BSW orientado ao *hardware* através do RTE, que é uma camada de abstração de *software*. No lado superior do RTE, essa camada permite o desenvolvimento de *software* competitivo e específico de OEMs. No seu lado inferior, permite a padronização e a independência do *software* produzido pelos OEMs. Outras características desta arquitetura são a escalabilidade do *software* de ECUs para várias linhas e variantes de carros, a possibilidade de distribuir aplicações (módulos de *software* funcionais) entre ECUs e a capacidade de integrar módulos de *software* de diferentes fontes.

O BSW é ainda dividido nas seguintes camadas: *Services Layer*, *ECU Abstraction Layer* e Microcontroller Abstraction Layer (MCAL). A separação do *Application Layer* do BSW, realizada pelo RTE, inclui o controlo da troca de

dados entre essas camadas. Isso forma a base para uma estrutura de *software* independente de *hardware* e orientada aos componentes, sendo assim os componentes de *software* unidades individuais. Devido à sua independência do *hardware*, é possível desenvolver componentes de *software* sem ter conhecimento específico do *hardware* usado, bem como realocar com flexibilidade os componentes existentes para ECUs durante o desenvolvimento [2].

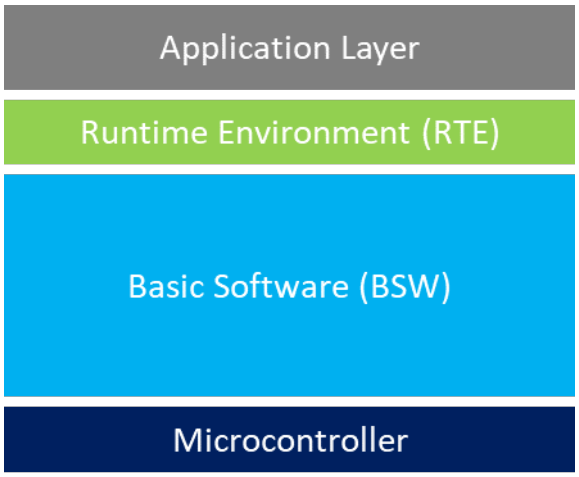


Figura 2.3: Visão geral da arquitetura em camadas[5]

2.2.1 Application Layer

A *Application Layer* é a camada superior da arquitetura e oferece suporte à implementação de funcionalidades personalizadas. Esta camada consiste em componentes de *software* específicos, que são um grupo de componentes de software AUTOSAR interconectados e executam tarefas específicas de acordo com as suas instruções. Cada elemento do software AUTOSAR possui parte da funcionalidade do software completo. A comunicação entre os componentes do software é possível através do uso de portos específicos usando o VFB, que também facilitam a comunicação entre os componentes e o BSW [5].

2.2.2 Runtime Environment

Como já foi mencionado anteriormente, esta camada fornece os serviços de comunicação entre os componentes de *software* e o BSW. A *Application Layer* consiste em vários componentes de *software* que são independentes e é através do RTE que estes comunicam entre si. A comunicação entre os componentes ocorre principalmente em 2 diferentes métodos, representados na figura 2.4: Cliente - Servidor, que permite ao cliente fazer um pedido de uma operação por um servidor que a suporta. O servidor executa a operação e fornece imediatamente

ao cliente o resultado (operação síncrona) ou então o cliente verifica por si mesmo a conclusão da operação (operação assíncrona) e Emissor - Recetor, que permite estabelecer uma comunicação tipicamente assíncrona, em que o emissor fornece dados que são exigidos por um ou mais recetores [5].

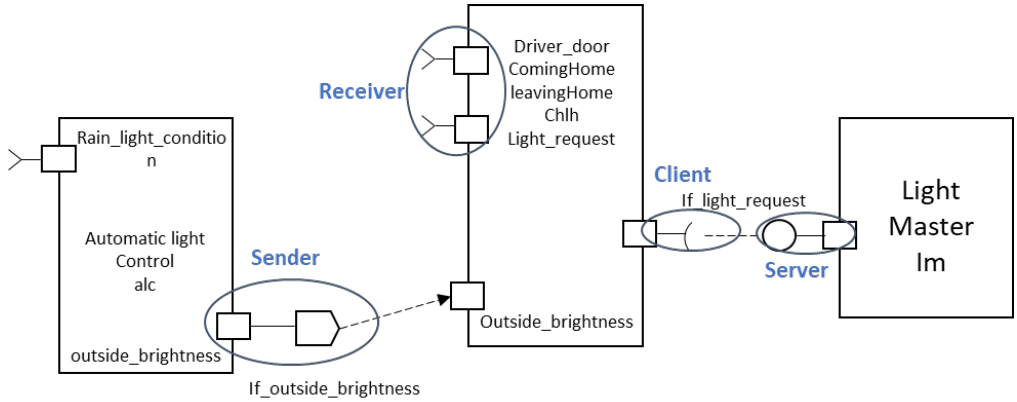


Figura 2.4: Visão geral das comunicações[5]

2.2.3 AUTOSAR Basic Software

Como já foi mencionado anteriormente, O BSW é ainda dividido em mais 3 camadas, representadas na figura 2.5: *Services Layer*, *ECU Abstraction Layer* e *Microcontroller Abstraction Layer*

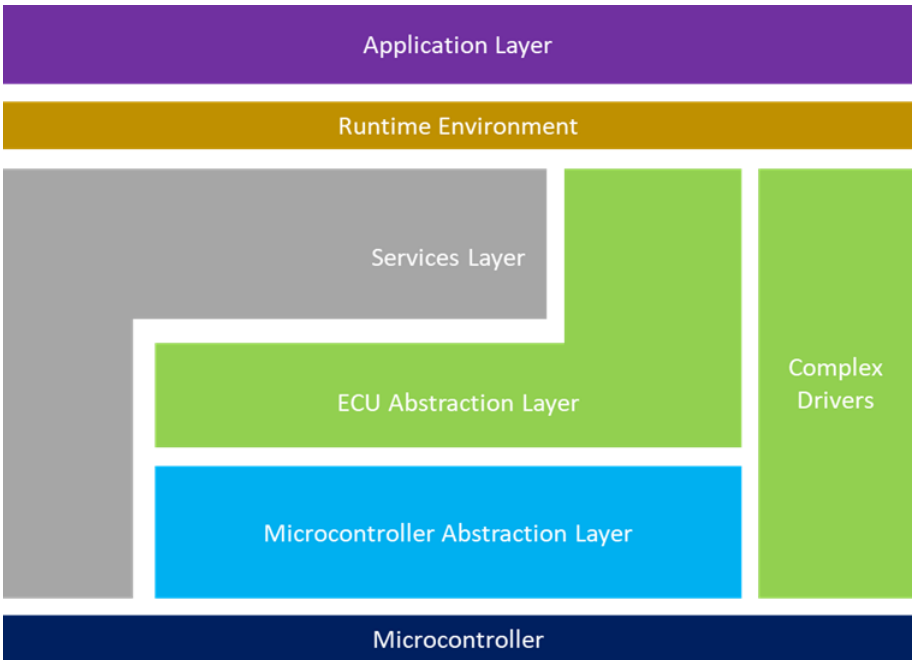


Figura 2.5: Visão geral do BSW [5]

2.2.3.1 Microcontroller Abstraction Layer

O MCAL é a camada mais baixa do BSW e contém drivers internos que são os módulos de software que têm acesso direto ao microcontrolador e periféricos internos.

2.2.3.2 ECU Abstraction Layer

Esta é a camada que fornece a interface para o acesso aos recursos dos ECUs, como a memória e portas I/O. Para além disso também fornece a interface para os drivers do MCAL.

2.2.3.3 Services Layer

A *Services Layer* é a camada superior do BSW. Fornece uma interface independente do microcontrolador e do *hardware* do ECU. O *layer* oferece vários serviços, tais como: funcionalidade do sistema operativo, serviços de comunicaçã, serviços de memória (gestão do NVRAM), serviços diagnósticos, como por exemplo controlo de erro etc, gestão do estado do ECU e monitorização lógica e temporal do *flow* do programa.

2.2.4 Complex Device Drivers Layer

Para além das 3 camadas principais, existe também o Complex Device Drivers (CDD), que se extende desde o *hardware* até ao RTE. O CDD cumpre funções especiais e requisitos de tempo necessários para operar sensores e atuadores complexos. Fornece a possibilidade de integrar funcionalidades para fins especiais. Para além disso, esta camada consiste de *drivers* para dispositivos que não são especificados no AUTOSAR [5].

2.3 AUTOSAR Adaptive Platform

Os carros têm continuado a transformarem-se em sistemas físicos cibernéticos - conectando-se à *internet* e a comunicar com *smartphones*. Os carros do futuro estarão conectados a quase tudo: casas inteligentes, infraestruturas das estradas e até mesmo a veículos no seu redor, tornando-se parte da Internet of Things (IoT), como representado na figura 2.6 [8].

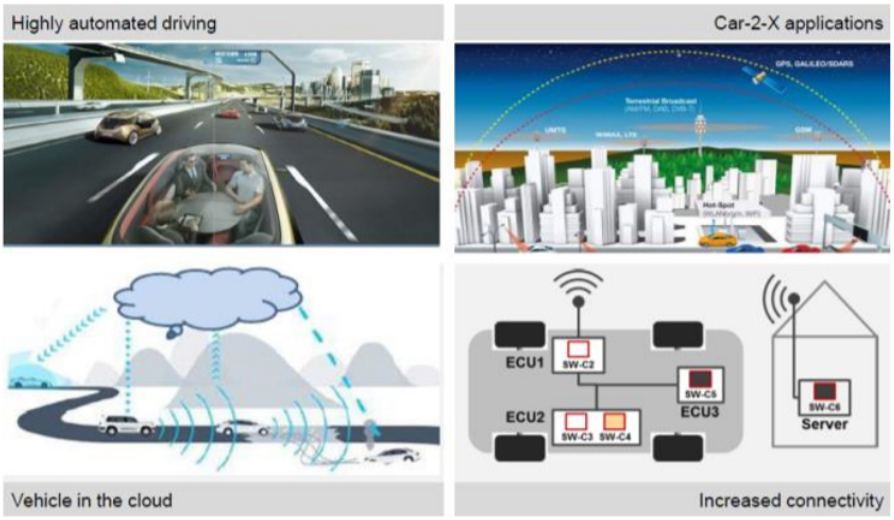


Figura 2.6: Casos do desenvolvimento contínuo da norma AUTOSAR [20]

Outra tendência para além do aumento da conectividade é a condução autónoma. Graças aos aprimoramentos dos sistemas de assistência ao motorista atuais, como o *Adaptive Cruise Control*, abriu-se o caminho para uma condução altamente automatizada. Estes novos recursos também adicionam novos requisitos à infraestrutura de software, que possui essas funcionalidades. Além dos requisitos existentes, tais como segurança e proteção, a arquitetura do software deve suportar também, por exemplo, hardware com maiores especificações, comunicação com sistemas de *back-end*, bem como confiabilidade para controlar veículos autónomos. Uma avaliação ao AUTOSAR mostrou que estes novos re-

quisitos não podem ser realizados pelas arquiteturas de *software* antigas, onde quase toda a comunicação interna do veículo é feita por meio de um controlador instalado de modo a cumprir os requisitos do OEM, como tempos de inicialização ou segurança funcional. Em geral, é necessária uma infraestrutura de *software* muito mais flexível, que seja capaz de se adaptar aos requisitos específicos das aplicações num determinado momento. Uma extensão da arquitetura de software inicial do AUTOSAR para sistemas profundamente embebidos acabou por não ser viável. Consequentemente, uma nova arquitetura foi criada com o objetivo de complementar a arquitetura existente [20].

2.3.1 Aspetos chave da nova arquitetura E/E

2.3.1.1 Integração de plataformas de software heterogéneas

Apesar dos ECUs de infoentretenimento usarem normalmente Linux ou sistemas operativos comerciais de uso geral, a AUTOSAR Classic platform era a norma para ECUs embebidos. Com os novos casos de uso e a crescente exigência de aplicações incorporadas, surgiu um terceiro tipo de ECUs com características diferentes que são interconectadas com as arquiteturas E/E existentes [20].

2.3.1.2 Comunicação orientada a serviços e baseada em sinais

A comunicação automóvel tradicional é baseada na ideia dos ECUs fornecer sinais para outros ECUs como transmissão. Este paradigma serve bem para dados de controlo de tamanho limitado, que devem ser comunicados ciclicamente. Aplicações avançadas, como direção altamente automatizada com maior exigência de carga útil, por exemplo, para trocar listas de comprimento dinâmico de objetos detetados por um conjunto de sensores e Ethernet como um sistema de comunicação requerem protocolos mais sofisticados. O conceito da comunicação orientada a serviços é baseada em aplicações que fornecem um serviço no sistema de comunicação e outras fazem *request* desse serviço. Os dados são enviados apenas para quem faz o *request* [20].

2.3.2 Arquitetura

A arquitetura da Adaptive Platform, representada na figura 2.7, tal como a Classic, é definida como uma arquitetura em camadas, embora com pequenas diferenças, mas também características únicas que a tornam adequada para os problemas e requisitos de uma nova era, principalmente por ser uma Service Oriented Architecture (SOA). Um modelo no qual os serviços são disponibilizados numa rede e as aplicações podem solicitar esses serviços de forma dinâmica e, portanto, "adaptativa" [10].

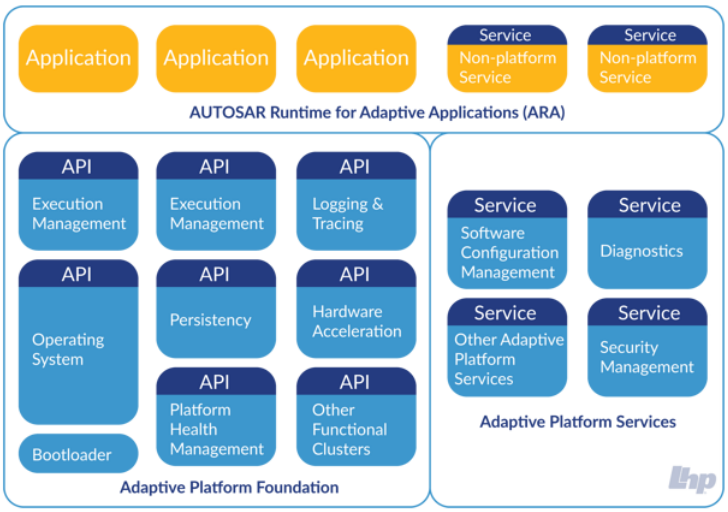


Figura 2.7: Visão geral da arquitetura da Adaptive Platform[10]

2.3.2.1 Adaptive AUTOSAR Applications

Na camada superior, podemos encontrar as novas Adaptive Applications (AA). As AAs, ao contrário dos componentes de software, são agora processos que podem ser simples ou *multithread*. Outra característica importante deles é que agora podem ser iniciados ou interrompidos a qualquer momento. Mesmo que um cenário semelhante possa ser feito no Classic usando partições, o aspeto chave está mais uma vez no comportamento estático do Classic ao contrário do dinâmico do Adaptive. As AAs podem ser ter que se atualizar todo o software, ao contrário da Classic Platform. Além disso, as AAs agora são operados por uma entidade familiar, mas nova, o AUTOSAR Runtime for Adaptive Applications (ARA). Este é composto por interfaces de aplicações provenientes dos blocos de construção da próxima camada, ou seja, os *Functional Clusters*. Além disso, esses clusters podem ser da Adaptive Platform Foundation ou dos Adaptive Platform Services. Outro aspeto fundamental do ARA é que, ao contrário do RTE, os clientes e serviços são vinculados dinamicamente em tempo de execução, no entanto, também permite opções para limitar o comportamento dinâmico, com o objetivo de diminuir o risco de efeitos indesejados. Por exemplo, algumas dessas opções incluem alocação dinâmica de memória apenas na fase de inicialização e até mesmo alocação fixa de processos para os núcleos do CPU [10].

2.3.2.2 Functional Clusters

O software da *Adaptive Foundation* e dos *Adaptive Services* é apresentado na forma de Functional Clusters. Simplificando, um *functional cluster* é um conjunto de requisitos agrupados pela funcionalidade a que se referem. São, de certa

forma, semelhantes ao conceito do BSW da *Classic Platform*. O objetivo principal é oferecer funcionalidades na forma de serviços à aplicação. No entanto, por fazerem parte da Adaptive Platform, e ao contrário do BSW, são processos que podem ser *single threaded* e *multithreaded*. Outra característica importante é a natureza dinâmica em vez da natureza estática tradicional do BSW. Assim como na *Classic Platform*, também há a necessidade de armazenamento não volátil, comunicação e diagnóstico etc, que são tratados pela memória não volátil, pelo Diagnostic Communications Manager (DCM) e pelo Diagnostic Event Manager (DEM). Estas necessidades são tratadas na *Classic Platform* pelos *Functional Clusters* [10].

2.3.3 Diferenças entre as plataformas Classic e Adaptive

O cenário atual dos sistemas automóveis E/E exige a cooperação das duas plataformas, visto que os requisitos rigorosos de tempo real de certas aplicações se enquadram perfeitamente nos ECUs embebidos cobertos pela *Classic Platform*, enquanto que os requisitos de *soft real-time* e de maiores especificações pertencem à *Adaptive Platform* [10]. As maiores diferenças entre estas duas arquiteturas são as seguintes:

- O *Classic* AUTOSAR foi feito para ser utilizado numa arquitetura de um ou vários núcleos, enquanto que o *Adaptive* AUTOSAR foi projetado para tirar proveito das arquiteturas High Performance Computing (HPC);

- O *Classic* AUTOSAR define o sistema operativo, enquanto que o *Adaptive* AUTOSAR define apenas um contexto de execução e uma interface do sistema operativo;

- O *Classic* AUTOSAR é de natureza estática, enquanto que o *Adaptive* AUTOSAR oferece uma dinâmica "planeada", tanto na implementação de aplicações como a nível de comunicação e recursos;

- O *Classic* AUTOSAR foi feito para ECUs profundamente embebidos, enquanto que o *Adaptive* AUTOSAR foi feito para a nova geração de ECUs.

Por fim, como já foi mencionado, o *Adaptive* AUTOSAR não foi feito para substituir o *Classic* AUTOSAR. Ambos devem coexistir e cooperar em vencer os desafios apresentados à indústria automóvel.

2.4 Protocolos de Comunicação para Veículos

2.4.1 CAN

O protocolo CAN é um método de comunicação entre dispositivos eletrónicos integrados num veículo, como os sistemas de gestão do motor, suspensão ativa, ar condicionado, airbags, etc. A ideia foi iniciada por *Robert Bosch* em 1983 para melhorar a qualidade e a segurança dos automóveis, aumentando a sua confiabilidade e a eficiência do combustível. O método da *Bosch*, lançado pela primeira vez em 1986, também proporcionou avanços na tecnologia de comunicação, o que foi significativo, visto que também houve progresso nas indústrias de eletrónica e semicondutores da época, o que levou a novas tecnologias, mas também a desafios para os engenheiros da indústria automóvel. Por exemplo, a eletrónica disponibilizou recursos mais complexos, incluindo a capacidade de comunicação entre dispositivos. Os engenheiros da indústria automóvel, frequentemente recebiam a tarefa de incorporar tais dispositivos, garantindo que funcionassem sem erros. O protocolo CAN simplificou o processo, pois permitiu que os diferentes módulos eletrónicos pudessem comunicar entre si através do uso dum cabo comum [18].

2.4.1.1 A necessidade do protocolo CAN

Um veículo contém uma rede de dispositivos eletrónicos que partilham dados e informação entre si. Um exemplo de comunicação entre dispositivos é a unidade de controlo de transmissão do automóvel. Este está programado para mudar automaticamente a mudança do veículo, tendo em conta a sua velocidade, utilizando os dados da unidade de controlo do motor e vários sensores no sistema, cada mecanismo possuindo um ECU. No entanto, para que dois ou mais dispositivos comuniquem entre si, devem estar equipados com o devido hardware e software. Antes do CAN ser usado em veículos, os dispositivos estavam conectados através de ligações ponto a ponto, o que era o suficiente devido às funções serem básicas. Mas à medida que a área da eletrónica foi avançando, tornou-se cada vez mais difícil a troca de dados entre os dispositivos. O protocolo CAN foi inventado para resolver este problema [18].

2.4.1.2 Como funciona a comunicação no protocolo CAN

O protocolo CAN é uma rede em que não há um controlo centralizado que dê acesso aos nós individuais para ler e escrever dados no barramento CAN. Quando um nó está pronto para transmitir dados, este verifica se o barramento está ocupado e escreve um frame na rede. Os frames CAN que são transmitidos não contêm os endereços do nó transmissor ou dos nós recetores pretendidos. Em vez disso, possuem um ID exclusivo que o permite identificar em toda a rede. Todos os nós da rede recebem o frame e, dependendo do seu ID, cada um irá decidir

se o aceita ou não. Se vários nós tentarem transmitir uma mensagem ao mesmo tempo, o nó com a prioridade mais alta (ID mais baixo) ganha automaticamente o acesso ao barramento. Os nós de prioridade mais baixa esperam até que o barramento fique disponível antes de voltar a tentar transmitir novamente [17].

2.4.1.3 Vantagens do protocolo CAN

O protocolo CAN fornece uma rede de baixo custo e estável que permite, como já foi mencionado, aos vários dispositivos comunicarem entre si. A vantagem disso é que permite aos ECUs ter apenas uma única interface CAN em vez de várias entradas analógicas e digitais para todos os dispositivos do sistema, o que diminui o custo geral e o peso dos automóveis, representado na figura 2.8. Por outro lado, cada um dos dispositivos possui um chip controlador CAN. Os dispositivos conseguem ver todas as mensagens transmitidas e cada um decide se uma certa mensagem é relevante ou se deve ser filtrada. Esta estrutura permite fazer modificações em redes CAN com o mínimo de impacto na mesma. Para além disso, como já foi mencionado, as mensagens possuem prioridade, ou seja, se dois nós tentarem enviar uma mensagem em simultâneo, o que tiver a prioridade mais alta será o que vai transmitir primeiro. Finalmente, o protocolo CAN inclui o Cyclic Redundancy Code (CRC) para realizar a verificação de erros no conteúdo de cada frame. Os frames com erros são desconsiderados pelos nós, e é transmitido um frame para sinalizar o erro para a rede. Os erros globais e locais são diferenciados pelo controlador e, se muitos erros forem detetados, os nós individuais podem parar de transmitir erros ou se desconectar completamente da rede [17].

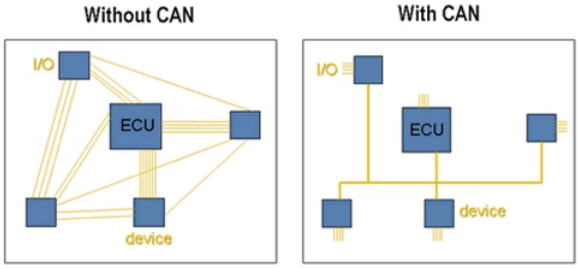


Figura 2.8: Fiação da rede com e sem CAN [17]

2.4.2 Ethernet

A Ethernet é a tecnologia tradicional para conectar dispositivos numa Local Area Network (LAN) ou Wide Area Network (WAN). Permite que os dispositivos comuniquem entre si por meio de um protocolo, que é um conjunto de regras ou linguagem de rede comum. Este protocolo descreve como os elementos da rede formatam e transmitem dados, de maneira a que os outros dispositivos na mesma

LAN ou WAN possam reconhecer, receber e processar a informação. Os dados percorrem o meio através de cabos Ethernet. Os dispositivos conectados que usam cabos para aceder a uma rede especificada geograficamente - em vez de uma conexão sem fios - provavelmente usam Ethernet. Desde empresas a jogadores de vídeo-jogos, são diversos os utilizadores que desfrutam dos benefícios da Ethernet, tais como confiabilidade e segurança. Em comparação com a tecnologia Wireless Local Area Network (WLAN), a Ethernet é normalmente menos vulnerável a perturbações, oferecendo um maior nível de segurança e controlo da rede do que a tecnologia sem fios, devido aos dispositivos serem conectados usando fiação física, o que dificulta o acesso de terceiros aos dados da rede [3].

2.4.2.1 Ethernet em AUTOSAR

Até há alguns anos atrás, CAN e LIN eram os únicos sistemas *bus* usados em automóveis. O desejo por uma maior largura de banda e maiores requisitos a nível da segurança, levaram ao desenvolvimento e introdução do FlexRay. Ao contrário do CAN, o FlexRay é um sistema *bus* complexo e caro. Devido a isso, o CAN ainda era utilizado no acesso externo no diagnóstico dos veículos. No entanto, o tempo necessário para programar ECUs aumentou drasticamente devido à largura de banda limitada do CAN e ao aumento da quantidade de conteúdo de software. O Diagnostics over Internet Protocol (DoIP) foi desenvolvido há vários anos para resolver este problema. Este protocolo foi o primeiro a ser baseado em Ethernet como tecnologia de rede no ambiente automóvel. A Ethernet oferece a vantagem de ter uma alta largura de banda, sendo o DoIP que lançou as bases para o uso da Ethernet nos automóveis.

No processo de carregamento, os veículos elétricos ou híbridos comunicam com o ponto de carregamento, sendo essa comunicação baseada em TCP/IPv6 e um protocolo dedicado Smart Charge Communication (SCC), a fim de trocar dados como o tipo de cobrança (AC/DC), data e hora da cobrança, duração da cobrança e informações sobre taxas e pagamentos. O cabo Ethernet era o que impedia o uso generalizado da tecnologia da rede em veículos. No entanto, a introdução da nova camada física *BroadR-Reach* tornou a opção da Ethernet viável também para a comunicação em veículos. O *BroadR-Reach* oferece uma largura de banda de 100 MBit/s, o que representa um aumento de 100 vezes na velocidade em comparação com o CAN, sem aumento nos custos na fiação. A Ethernet em combinação com o protocolo da Internet, o Transmission Control Protocol (TCP) e o User Datagram Protocol (UDP) permitem a transição de um esquema de comunicação orientado a dados para um esquema de comunicação orientado a serviços [27].

A Ethernet faz parte do AUTOSAR desde a versão 4.0. Ao contrário das outras comunicações, a Ethernet exibe uma série de aspetos especiais - que se

relacionam com as camadas de protocolo superiores - Internet, UDP e TCP em particular. Enquanto que as interfaces para CAN, LIN e FlexRay implementam a interface AUTOSAR Protocol Data Unit (PDU) diretamente, a interface Ethernet encaminha os dados para a *stack* TCP/IP ou recebe dados dela. Os protocolos IP, UDP e TCP são processados na *stack* TCP/IP. Para isso, é utilizada uma *stack off-the-shelf* TCP/IP. O paradigma no qual a família de protocolos TCP/IP se baseia é a utilização de *sockets*. Um *socket* é unicamente identificado pelo de endereço IP e pela porta dos nós finais remotos e locais. Através de um *socket*, os dados do utilizador são encaminhados da *stack* TCP/IP para a aplicação ou vice-versa. Este paradigma era incompatível com o conceito PDU do AUTOSAR. A transformação da comunicação baseada em *sockets* em comunicação baseada no PDU e vice-versa é a tarefa do Socket Adaptor Module (SoAd). Fornece a interface PDU familiar aos módulos do nível superior, que integra totalmente a *stack* Ethernet na arquitectura AUTOSAR. A *stack* Ethernet especificada no AUTOSAR 4.0 estabeleceu uma base para a recepção e envio de PDUs por Ethernet, considerando o caso de uso do DoIP [27].

2.5 ARXML Files

Um ficheiro ARXML é um ficheiro de configuração guardado em formato AUTOSAR XML (ARXML). Estes ficheiros contêm informação de configuração e especificação em formato XML de um ECU. O formato ARXML foi desenvolvido para padronizar o intercâmbio de dados entre parceiros de desenvolvimento de software automóvel. Os ficheiros ARXML são criados pelo produtor automóvel original, o que significa que a informação exata em cada ficheiro ARXML pode variar [11].

XML é uma linguagem criada pelo World Wide Web Consortium (W3C) para definir uma sintaxe para a codificação de documentos. Isto é feito através da utilização de *tags* que definem a estrutura do documento, assim como a forma como o documento deve ser armazenado e transportado. Pode ser comparada com outra linguagem *markup* - a linguagem HTML usada para codificar páginas web. O HTML utiliza um conjunto predefinido de símbolos *markup* que descrevem o formato do conteúdo de uma página web. O que diferencia o XML, no entanto, é que é extensível. O XML não tem uma linguagem *markup* pré-definida, como o HTML tem. Em vez disso, o XML permite aos utilizadores criar os seus próprios símbolos para descrever o conteúdo, fazendo um conjunto ilimitado e auto-definido de símbolos. Essencialmente, o HTML é uma linguagem que se concentra na apresentação de conteúdos, enquanto que o XML é uma linguagem dedicada à descrição de dados utilizada para armazenar dados [9].

Os caracteres que compõem um documento XML são divididos em *markup* e conteúdo, que podem ser distinguidos por um parser através de regras sintácticas

simples. As *tags* de *markup* começam com o carácter < e terminam com >. Todo o texto que não esteja *tagged* é conteúdo. Existem três tipos de *tags*:

- Start-tag: <section>;
- End-tag: </section>;
- Empty-element tag: <line-break/>.

Um nó é um componente lógico do documento que começa com uma start-tag e termina com uma end-tag correspondente ou que consiste apenas duma empty-element tag. Os caracteres entre as tags, se existirem, são o conteúdo do nó, e podem conter *markup* também, incluindo outros nós, que são chamados de nós *child*.

Um atributo é outro componente lógico que consiste num par nome-valor que existe dentro de uma start-tag ou de empty-element tag. Um exemplo é , onde os nomes dos atributos são ”src”e ”alt”, e os seus valores são ”madonna.jpg”e ”Madonna”respetivamente. Um atributo XML só pode ter um único valor e cada atributo pode aparecer, no máximo, uma vez em cada nó [26].

Capítulo 3

Investigação de Ferramentas a Utilizar

3.1 Bibliotecas C++

Visto que a framework está escrita em C++, optou-se por utilizar essa linguagem também para o programa. Sendo então necessário escolher uma biblioteca *license free* que permitisse fazer o parsing de ficheiros XML em C++. 3 das bibliotecas mais utilizadas para este fim são: Xerces, Pugixml e Rapidxml [23].

3.1.1 Xerces

Xerces C++ é uma biblioteca de XML parsing C++ que dá a capacidade de ler e escrever dados em ficheiros XML às aplicações. Esta biblioteca foi projetada para analisar, gerar, manipular e validar ficheiros XML usando as APIs: DOM, SAX e SAX2. O Xerces C++ é fiel à recomendação XML 1.0 e muitos padrões associados. O parser proporciona alto desempenho, modularidade e escalabilidade. O *source code*, *samples* e documentação API são fornecidos com o parser. Para a portabilidade, tiveram o cuidado de ter um uso mínimo de *templates* e um uso mínimo de ifdefs [25].

O parser é utilizado principalmente para:

- Construção de servidores Web XML-savvy;
- Construir aplicações que utilizam XML como formato de dados;
- Validação em tempo real para a criação de editores XML;
- Assegurar a integridade dos dados de *e-business* expressos em XML[25].

No entanto, esta biblioteca não é *license free*.

3.1.2 Pugixml

PugiXml é outra biblioteca *open source* C++ que realiza um *parse* de ficheiros XML rápido e eficaz [6]. As principais *features* desta biblioteca são:

- Interface do tipo DOM com capacidades de travessia/modificação;
- Parser XML extremamente rápido que constrói a DOM *tree* a partir de um ficheiro/buffer XML;
- Implementação de XPath 1.0 para consultas complexas de árvores orientadas por dados;
- Suporte completo Unicode com variantes de interface Unicode e conversão automática de codificação [13].

Esta biblioteca foi desenvolvida em 2006 e é atualizada desde então. Todo o código é distribuído sob a licença do MIT, tornando-o completamente livre para utilização tanto em aplicações *open source* como em aplicações proprietárias. A versão mais recente é a versão 1.11, lançada em Novembro de 2020 [13].

3.1.3 Rapidxml

RapidXml é uma tentativa de criar o parser DOM XML mais rápido possível, mantendo ao mesmo tempo a facilidade de utilização e portabilidade. É um parser escrito em C++, com velocidade de parsing próxima da função strlen() executada sobre os mesmos dados. A biblioteca inteira está contida num único ficheiro *header*. Para o utilizar basta copiar o ficheiro rapidxml.hpp para um local conveniente, e incluí-lo onde for necessário. O RapidXml não tem outras dependências além de um subconjunto muito pequeno de bibliotecas *standard* C++ (<cassert>, <cstdlib>, <new> e <exception>). Dá parse com sucesso a strings wchar-t contendo UTF-16 ou UTF-32. UTF-8 também é totalmente suportado, incluindo todas as referências de carateres numéricos, que são expandidas em sequências de bytes UTF-8 apropriadas. Utiliza um *memory pool object* especial para alocar nós e atributos, porque a alocação direta utilizando um novo operador seria demasiado lenta. Faz o parsing e produz com sucesso árvores completas de todos os ficheiros XML válidos no conjunto de conformidade W3C (mais de 1000 ficheiros especialmente concebidos para encontrar falhas nos processadores XML). No modo destrutivo, realiza a normalização do espaço em branco e a substituição de entidades de carácter por um pequeno conjunto de entidades incorporadas. Por outro lado a API é minimalista, para reduzir ao máximo o tamanho do código, e facilitar a utilização em ambientes embebidos [12].

O RapidXml atinge a sua velocidade através da utilização de várias técnicas:

- Parsing in-situ: ao construir a árvore DOM, o RapidXml não faz cópias de dados de strings, tais como nomes e valores de nós, em vez disso, armazena apontadores para o interior do texto de origem;
- Utilização de técnicas de metaprogramação de *templates*. Através do uso dos *templates*, o compilador C++ gera uma cópia separada do código de parse para qualquer combinação de parser *flags* utilizadas. Em cada cópia, todas as decisões possíveis são tomadas no *compile time* e todo o código não utilizado é omitido;
- Utilização extensiva de*lookup tables* para parsing [12].

Isto resulta num código pequeno e rápido: um parser que é personalizado de acordo com as necessidades exatas de cada utilização, e por ser *license free*, decidiu-se optar por esta biblioteca.

3.2 GUI

3.2.1 Qt

Neste projeto, o *script* desenvolvido é integrado numa *framework* desenvolvida pela CES, que foi desenvolvida com Qt. A Qt é uma *open source framework* usada no desenvolvimento de interfaces gráficas na linguagem de C++, possuindo suporte de multimédia, gráficos 2d e 3d, incluindo animações de imagem, e outras funcionalidades para ajudar no trabalho com sql, xml, *unit test* e suporte para outras linguagens. Para além disso, a Qt oferece recursos para reproduzir ficheiros de áudio e vídeo. Por outro lado, a Qt é também muito útil no processo de desenvolvimento, pois é possível executar e dar *debug* no ambiente de desenvolvimento antes de se transpor para a plataforma de destino, permitindo a validação antecipada da interface, o que melhora significavelmente o tempo de desenvolvimento. Outra característica única do Qt é o conceito de sinais e *slots* (onde um *widget* ativa o sinal que está conectado ao *slot* (função a ser executada) [15]. A primeira versão do Qt foi lançada a 20 de maio de 1995 e a versão mais recente (6.1) foi lançada a 6 de maio de 2021 [14].

A Qt suporta vários compiladores, incluindo o compilador GCC C++, a suite Visual Studio e PHP através de uma extensão para PHP5. Por outro lado, também fornece a *feature* Qt Quick, que inclui uma linguagem de scripting declarativa chamada QML, que permite utilizar JavaScript. Com esta *feature*, tornou-se possível o desenvolvimento rápido de aplicações para dispositivos móveis [16]. Algumas aplicações que usam Qt são, por exemplo: Ableton Live (um dos*softwares* de produção musical mais usados na indústria), EAGLE (*software* EDA com

captura esquemática, layout de PCBs, *auto-router* e recursos CAM) e Roblox (videojogo popular).

3.2.2 WxWidgets

Outra alternativa seria o WxWidgets. Criado em 1992 por Julian Smart na Universidade de Edimburgo, inicialmente era um projeto com o fim de criar aplicações para sistemas Unix e Windows, crescendo para suportar MacOS, entre outros conjuntos de ferramentas e plataformas. O número de programadores que contribuem para o projeto está agora entre as centenas e o kit de ferramentas tem uma grande base de utilizadores, desde programadores *open source* a empresas. O WxWidgets fornece uma API fácil de usar para criar GUIs em múltiplas plataformas que utilizam os controlos da plataforma nativa. As principais *features* do WxWidgets são: ajuda online, *network programming*, *streams*, *clipboard*, *drag and drop*, *multithreading*, carregamento e gravação de imagens numa variedade de formatos populares, visualização e impressão de HTML, entre outras. Embora o WxWidgets seja escrito em C++, pode ser utilizado noutras linguagens incluindo Python, Perl, e C [28]. A sua última versão foi lançada a 14 de abril de 2021 [29].

Algumas aplicações criadas utilizando o WxWidgets são: Code::Blocks (C/C++ IDE), KiCad (um conjunto de software gratuito para automatização de desenho eletrónico) e BitTorrent (aplicação de partilha de ficheiros peer-to-peer).

Capítulo 4

Desenvolvimento do Projeto

4.1 Informação a ser extraída dos ficheiros ARXML

O objetivo do parser é obter informação sobre o ECU em questão através dos CAN frames. Sendo assim, a informação procurada nos ficheiros ARXML está dividida em 3. Um frame vai ser composto pelas variáveis: o seu nome, o seu id e um vetor que contém um conjunto de PDUS. Nesse conjunto de PDUS, cada um deles vai possuir as variáveis: o seu nome e um vetor que contém um conjunto de ISignals. Nesse conjunto de ISignals, cada um deles vai possuir as variáveis: o seu nome, signal length e a sua descrição. Ou seja, cada frame vai possuir um conjunto de PDUs, e cada PDU vai possuir um conjunto de ISignals, representado na figura 4.1.

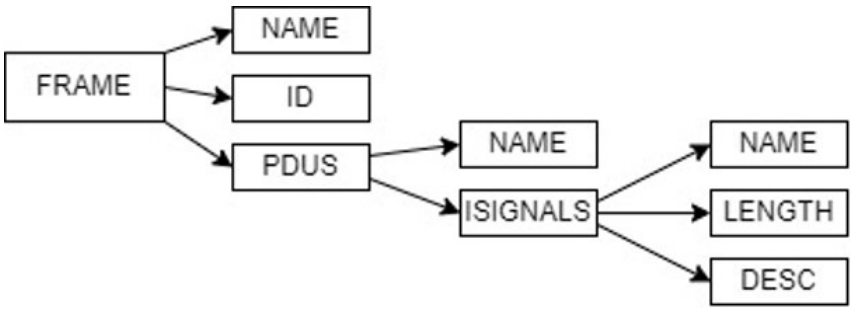


Figura 4.1: Estrutura dos dados a obter

Nas figuras 4.2 a 4.4 estão os class diagrams das diferentes classes criadas onde está guardada a informação obtida. Os métodos destas classes funcionam apenas como getters, são chamados no código apenas para aceder às variáveis.

Nas figuras 4.5 a 4.10, estão representados os ficheiros .h e .cpp de cada uma das classes.

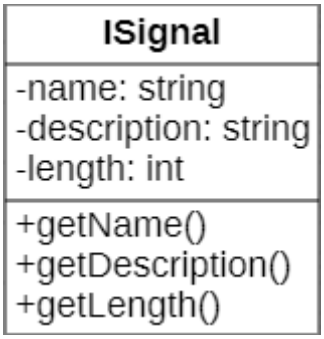


Figura 4.2: Class diagram da class ISignal

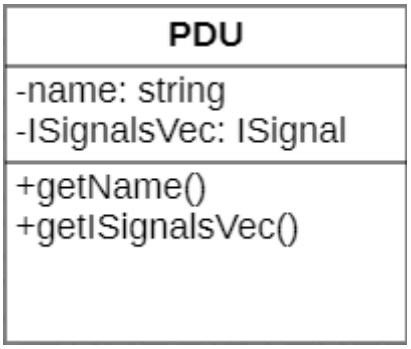


Figura 4.3: Class diagram da class PDU

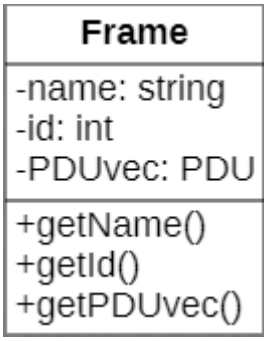


Figura 4.4: Class diagram da class Frames

```
class ISignal {

private:

    std::string name, descEn, signal_length;

public:

    std::string getSignalName(void);
    std::string getSignalDesc(void);
    std::string getSignalLength(void);
    ISignal(std::string n, std::string desc, std::string sig_length);
    ~ISignal();
};
```

Figura 4.5: Ficheiro .h da class ISignal

```
ISignal::ISignal(std::string n, std::string desc, std::string sig_length) {

    name = n;//
    descEn = desc;//
    signal_length = sig_length;//
}

ISignal::~ISignal(){

}

std::string ISignal::getSignalName(void) {

    return name;
}

std::string ISignal::getSignalDesc(void) {

    return descEn;
}

std::string ISignal::getSignalLength(void){

    return signal_length;
}
```

Figura 4.6: Ficheiro .cpp da class ISignal

```
class PDU
{

private:

    std::string name;
    std::vector<ISignal> signals;

public:

    PDU(std::string n, std::vector<ISignal> ISignalsvec);
    std::string getPDUName(void);
    std::vector<ISignal> getPDUsignals(void);
    ~PDU();
};
```

Figura 4.7: Ficheiro .h da class PDU

```
PDU::PDU(std::string n, std::vector<ISignal> ISignalsvec){

    name = n;
    signals = ISignalsvec;
}


PDU::~PDU()
{
}

std::string PDU::getPDUName(void) {

    return name;
}

std::vector<ISignal> PDU::getPDUsignals(void) {

    return signals;
}
```

Figura 4.8: Ficheiro .cpp da class PDU

```
class Frame{

private:
    //nome, id, pdus
    std::string id, name;
    std::vector<PDU> PDUS;

public:

    Frame(std::string uid, std::string n, std::vector<PDU> PDUSvec);
    std::string getFrameId(void);
    std::string getFrameName(void);
    std::vector<PDU> getFramePDUS(void);
    ~Frame();
};
```

Figura 4.9: Ficheiro .h da class Frame

```
Frame::Frame(std::string uid, std::string n, std::vector<PDU> PDUSvec){

    id = uid;
    name = n;
    PDUS = PDUSvec;
}

Frame::~Frame(){

}

std::string Frame::getFrameId(void) {

    return id;
}

std::string Frame::getFrameName(void) {

    return name;
}

std::vector<PDU> Frame::getFramePDUS(void) {

    return PDUS;
}
```

Figura 4.10: Ficheiro .cpp da class Frame

4.2 Funcionamento do Parser

O main loop do programa está representado na figura 4.12, com o seu respetivo fluxograma geral de funcionamento na figura 4.11.

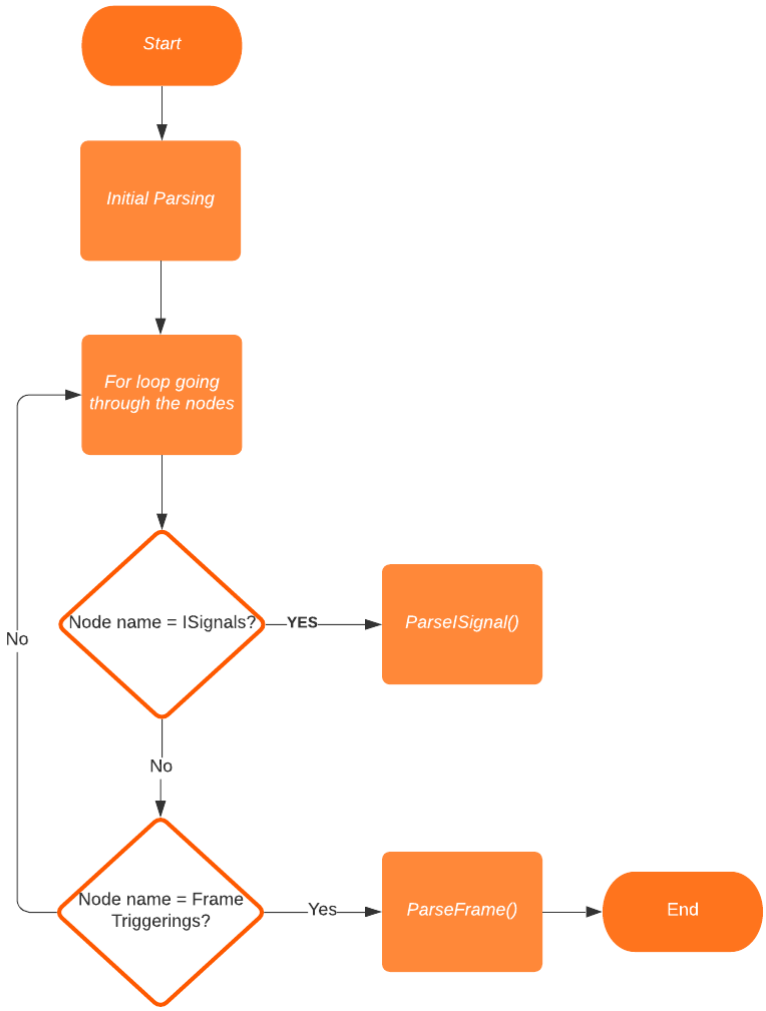


Figura 4.11: Fluxograma do funcionamento geral

```
void ParseArxml(vector<PDU> *PDUS, vector<ISignal> *signals, vector<Frame> *frames){

    //BEGINNING OF PARSE PROCESS //Parse the buffer using the xml file parsing library into doc
    xml_document<> doc;
    ifstream theFile("D:/VisualStudio/ArxmlParser/Ecu_Extract.arxml");
    vector<char> buffer((istreambuf_iterator<char>(theFile)), istreambuf_iterator<char>()); //this buffer contains the file content
    buffer.push_back('\0');
    doc.parse<0>(&buffer[0]);

    //INITIAL PARSE TO FIND THE ROOT NODE
    xml_node<> * root_node = doc.first_node("AUTOSAR");

    for(xml_node<> * ar_package_node = RootNode(root_node); ar_package_node; ar_package_node = ar_package_node->next_sibling()){

        xml_node<> * ar_package_short_name = ar_package_node->first_node("SHORT-NAME"); //Getting the packages' names in order to fin

        if(CheckIfISignalName(ar_package_short_name) == 0){ //if the comparison equals 0, it means we've found the package with the I

            ParseSignal(ar_package_node, PDUS, signals);
        }

        if(CheckIfFrameName(ar_package_short_name) == 0){ //if the comparison equals 0, it means we've found the package with the fra

            ParseFrame(ar_package_node, PDUS, frames);
        }
    }
}
```

Figura 4.12: Main loop do programa

Primeiramente o conteúdo do ficheiro ARXML é guardado num buffer, de modo a se poder utilizar a função parse(), fornecida pela biblioteca RapidXML, com o valor 0 que significa o uso default da função. Em seguida o programa faz o parse inicial do ficheiro até chegar ao nó onde se encontram os nós child que contêm os diferentes tipos de informação acerca do ECU em questão, representado na figura 4.13. Através do loop for representado na figura 4.11 e com a utilização da função next->sibling(), fornecida pela biblioteca RapidXML, o programa percorre os nós child desse nó, verificando um a um através dos ifs até encontrar um nó de interesse (ISignals ou Communication Clusters neste caso). Quando o programa deteta um desses nós, corre as funções ParseSignal ou ParseFrame respetivamente.

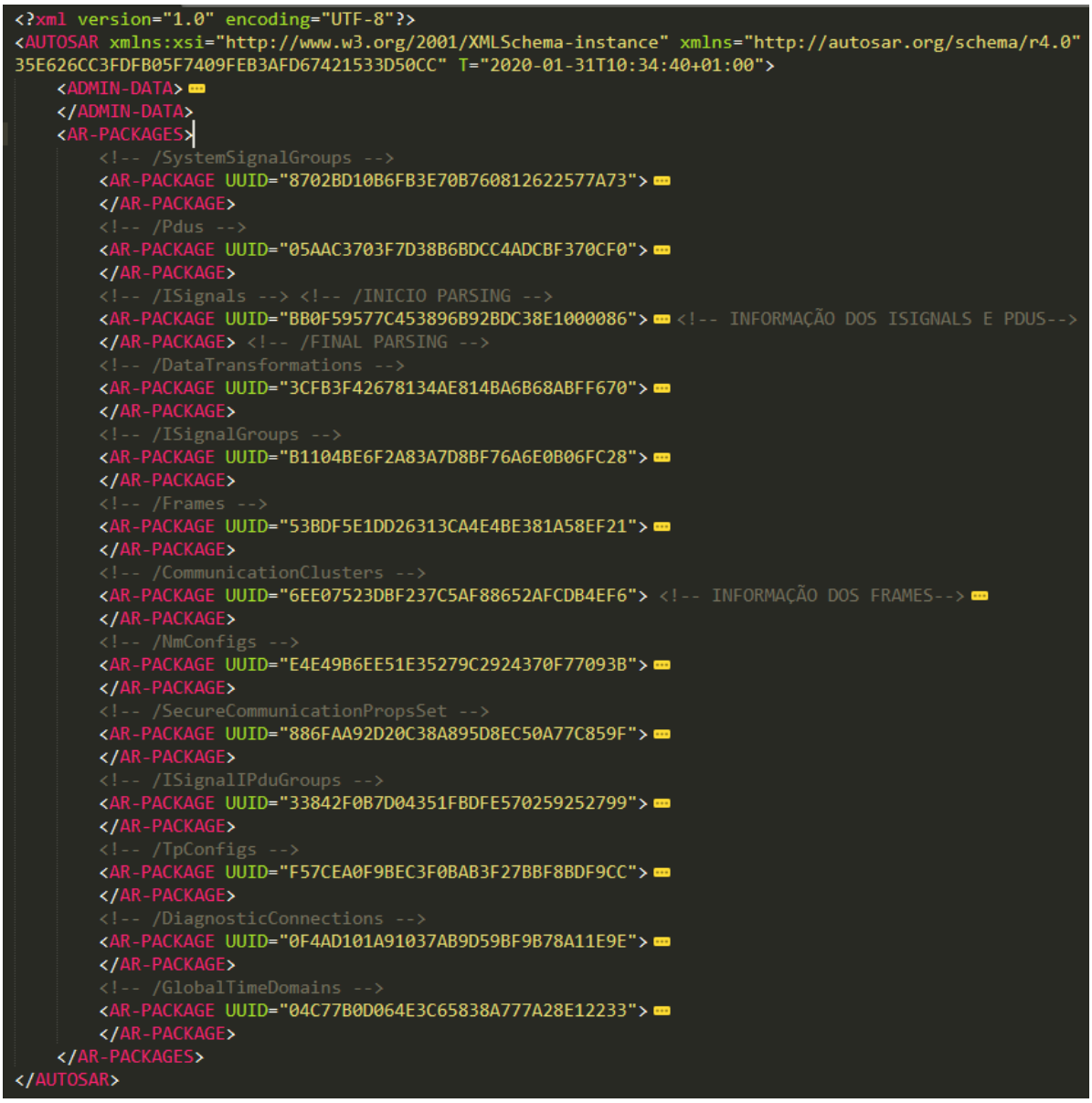


Figura 4.13: Organização dos nós do ficheiro ARXML

A informação dos PDUs e dos ISignals está presente em mais do que um nó nos ficheiros ARXML, no entanto, com a abordagem escolhida na elaboração deste programa, vai-se ao nó identificado pelo nome "ISignals"(o nome está guardado no valor dum dos nós child, neste caso o primeiro), representado na figura 4.14, onde se encontram todos os PDUs distribuídos por nós, representado na figura 4.15, tendo como nós child os diferentes ISignals pertencentes a cada PDU, representado na figura 4.16. Na figura 4.15 pode-se verificar que o primeiro nó child do nó dos PDUs possui o seu nome como valor, sendo aí onde o programa

vai buscar o nome de cada PDU e na figura 4.16 o mesmo se pode observar mas para o ISignal. Nas figuras 4.17 e 4.18 estão representados os nós com os valores da descrição e signal length dos ISignals, respetivamente.

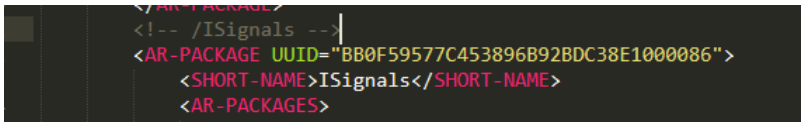


Figura 4.14: nó com o nome

Figura 4.15: Distribuição dos diferentes PDUs no ficheiro ARXML

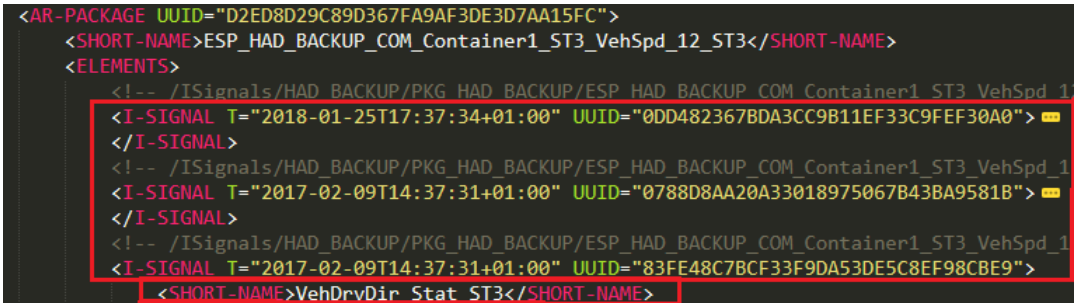


Figura 4.16: Distribuição dos diferentes ISignals dentro do nó do respetivo PDU

```
<DESC>
    <L-2 L="DE">Fahrtrichtung des Fahrzeugs</L-2>
    <L-2 L="EN">Vehicle driving direction state</L-2>
</DESC>
```

Figura 4.17: nó com a descrição do ISignal

```
<LENGTH>2</LENGTH>
```

Figura 4.18: nó com o signal length

Ou seja, a função ParseSignal (representada na figura 4.19) é responsável por ir buscar a informação tanto dos PDUs como dos ISignals. O primeiro loop for é o que permite ao programa percorrer os nós dos diferentes PDUs, guardando numa variável local o seu nome. Depois, de modo a se percorrer todos os nós contendo os seus diferentes ISignals, é-se utilizado outro loop for, guardando a informação de cada um e colocando-se cada ISignal num vetor dessa variável. Quando o programa acaba de percorrer todos os nós de ISignals de um certo PDU, é se colocada a informação desse PDU num vetor dessa variável e de seguida esvaziando-se o vetor de ISignals utilizado para guardar os ISignals do PDU acabado de ler, de modo a se poder reutilizar esse vetor para o próximo PDU a ser lido. O programa sai desta função quando se lerem todos os PDUs do ficheiro.

```
void ParseSignal(xml_node<> * ar_package_node, vector<PDU> *PDUS, vector<ISignal> *signals){

    for (xml_node<> * ar_package_pdu = PackageNode(ar_package_node); ar_package_pdu; ar_package_pdu = ar_package_pdu->next_sibling()) {

        string pdu_name = GetPDUName(ar_package_pdu);//save pdu name

        // Iterate over the signals
        for (xml_node<> * i_signal_node = ISignalNode(ar_package_pdu); i_signal_node; i_signal_node = i_signal_node->next_sibling()) {/

            //Signal name
            string signal_name = GetSignalName(i_signal_node);

            //Description
            //string descEn = "";
            string descEn = GetSignalDesc(i_signal_node);

            //Signal length
            string sig_length = GetSignalLength(i_signal_node);

            //Save the signal content into the object
            ISignal new_signal(signal_name, descEn, sig_length);//create an isignal variable
            signals->push_back(new_signal);//store the read isignal in the signals vector
        }

        PDU new_pdu(pdu_name, *signals);//create a pdu variable with the pdu name and its isignals
        PDUS->push_back(new_pdu);//store the read pdu in the pdus vector
        signals->clear();//clear the isignals vector in order to store the isignals from the next pdu
    }
}
```

Figura 4.19: Função ParseSignal

A informação dos frames está presente em mais do que um nó nos ficheiros

ARXML, tal como os ISignals e os PDUs, no entanto, com a abordagem escolhida na elaboração deste programa, o programa vai buscar os frames ao nó identificado pelo nome "Communication Clusters", onde estão listados todos os frames do ficheiro. Na figura 4.20 está representado um exemplo de um frame, podendo-se observar o nome e o ID respetivo destacados. Na função ParseFrame (representada na figura 4.21), de forma semelhante à função ParseSignal, é-se utilizado um loop for para se percorrer os nós que possuem os frames. No entanto, com esta abordagem, os PDUs pertencentes aos seus respetivos frames não estam nesta lista. Então, de modo a se contornar este obstáculo, depois de se guardar o nome e o ID de um frame, é feita a comparação do nome desse frame com todos os PDUs guardados da função anterior, de modo a se verificar quais é que pertencem a esse frame. Isso é possível pois os PDUs partilham o mesmo nome que os frames, apenas com mais especificações, daí se comparar os nomes apenas até à length do nome dos frames. Depois do programa verificar quais os PDUs que coincidem com o frame que está a ser lido, cria-se uma variável da class frame e guarda-se no vetor que guarda todos os frames lidos do programa.

```
<CAN-FRAME-TRIGGERING UUID="0A6B7B66240B3BBEADD54E5C8F06DD0B">
    <SHORT-NAME>ESP_HAD_BACKUP_COM_Container1_ST3_m7h6933xsls2hbt3g5pz7rbf</SHORT-NAME>
    <FRAME-PORT-REFS>
        <FRAME-PORT-REF DEST="FRAME-PORT">/EcuInstances/MPC_R/MPC_R_HAD_BACKUP_2k8epbkkrp1ckx3hmcel286xw/
        XDIS_17exblry01udpeh99lexcn8uc</FRAME-PORT-REF>
    </FRAME-PORT-REFS>
    <FRAME-REF DEST="CAN-FRAME">/Frames/HAD_BACKUP/PKG_HAD_BACKUP/ESP_HAD_BACKUP_COM_Container1_ST3</FRAME-REF>
    <PDU-TRIGGERINGS>
        <PDU-TRIGGERING-REF-CONDITIONAL>
            <PDU-TRIGGERING-REF DEST="PDU-TRIGGERING">/CommunicationClusters/HAD_BACKUP/HAD_BACKUP/HAD_BACKUP/
            ESP_HAD_BACKUP_COM_Container1_ST3_9wgr7acccdjxfn4xsmzzlmfj6</PDU-TRIGGERING-REF>
        </PDU-TRIGGERING-REF-CONDITIONAL>
    </PDU-TRIGGERINGS>
    <CAN-ADDRESSING-MODE>STANDARD</CAN-ADDRESSING-MODE>
    <CAN-FRAME-RX-BEHAVIOR>CAN-FD</CAN-FRAME-RX-BEHAVIOR>
    <CAN-FRAME-TX-BEHAVIOR>CAN-FD</CAN-FRAME-TX-BEHAVIOR>
    <IDENTIFIER>295</IDENTIFIER>
</CAN-FRAME-TRIGGERING>
```

Figura 4.20: Distribuição dos valores de um frame

```
void ParseFrame(xml_node<> * ar_package_node, vector<PDU> *PDUS, vector<Frame> *frames){

    for (xml_node<> * can_frame_trigg_node = FramePackageNode(ar_package_node); can_frame_trigg_node; can_frame_trigg_node = can_frame_trigg_node->next_sibling()){

        string frame_name = GetFrameName(can_frame_trigg_node);
        string frame_id = GetFrameId(can_frame_trigg_node);
        vector<PDU> pdu_temp;//temporary pdu vector that will store pdus that belong to a certain frame

        for (auto& it : *PDUS){//go through all the pdus that were parsed in the ParseSignal function

            string pdu_name_temp = it.getPDUName();//get the PDU name

            if (pdu_name_temp.compare(0, frame_name.length(), frame_name) == 0) {//check which PDUS belong to the current frame

                pdu_temp.push_back(it);//storing the pdus that belong to a certain frame in the pdu variable
            }
        }

        Frame new_frame(frame_id, frame_name, pdu_temp);//creating a frame variable with the read content
        frames->push_back(new_frame);//store read frame in the frames vector
    }

    PDUS->clear();
}
```

Figura 4.21: Função ParseFrame

Na figura 4.22 está representada a função utilizada para se ir buscar o ID do frame, de modo a se usar como exemplo para mostrar como o programa se comporta a ir buscar os valores, visto que as outras funções são semelhantes a esta. Dentro do nó dos frames, o programa navega para o nó child ”IDENTIFIER”, cujo valor será o ID do frame correspondente.

```
string GetFrameId(xml_node<> * can_frame_trigg_node) {

    string id;
    xml_node<> * frame_id = can_frame_trigg_node->first_node("IDENTIFIER");
    id = frame_id->value();
    return id;
}
```

Figura 4.22: Função GetFrameID

4.3 Integração do programa na Framework

A framework já possuía a feature de fazer o parsing de ficheiros DBC, que são outro tipo de base de dados. Sendo assim, foram feitas alterações ao source code da framework para que esta permitisse fazer também o upload de ficheiros ARXML, representado na figura 4.23, detetasse qual o tipo de base de dados que foi uploaded e corresse o programa respetivo ao tipo de ficheiro, representado na figura 4.24, sendo também feitas alterações ao programa desenvolvido para que este conseguisse abrir o ficheiro através do seu path. Foram também integrados os ficheiros das classes, mas devido a motivos de confidencialidade, não é possível partilhar o resto das alterações.

```
void QCanWidget::onChangeDbcClicked()
{
    QFileDialog dialog(this, "Select a database (ARXML/DBC File)", QString(), "DBC files (*.dbc *.arxml)");
```

Figura 4.23: Alterações para o upload de ficheiros ARXML

```
void QCanDecoder::loadDbc(const QString& dbcPath)
{
    if (dbcPath.contains(".arxml")){


        ArxmlParser m_ArxmlParser;
        // Parse arxml file and update list of CAN message descriptions
        m_canMessageDescriptions = m_ArxmlParser.readParserFile(dbcPath);
    }

    else if (dbcPath.contains(".dbc")){

        // Parse dbc file and update list of CAN message descriptions
        m_canMessageDescriptions = DbcParser::readParserFile(dbcPath);
    }

}
```

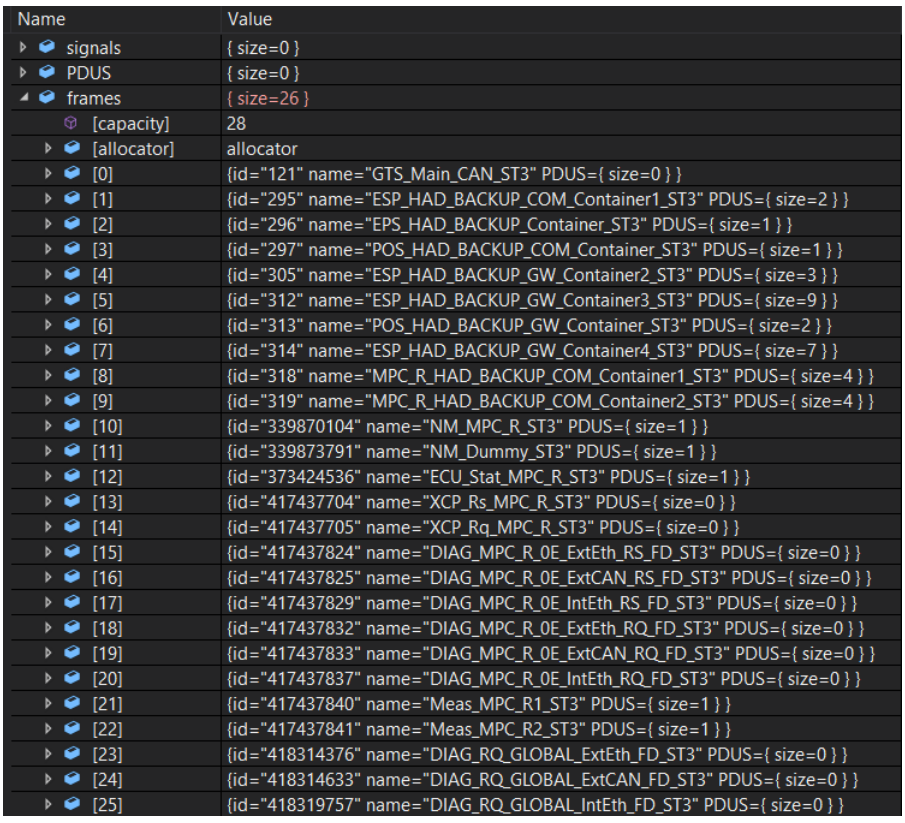
Figura 4.24: Função loadDbc com alterações

Capítulo 5

Resultados

5.1 Resultados do Parser

O estado dos vetores quando se corre o script desenvolvido está representado na figura 5.1.



Name	Value
signals	{ size=0 }
PDUS	{ size=0 }
frames	{ size=26 }
[capacity]	28
[allocator]	allocator
[0]	{id="121" name="GTS_Main_CAN_ST3" PDUS={ size=0 } }
[1]	{id="295" name="ESP_HAD_BACKUP_COM_Container1_ST3" PDUS={ size=2 } }
[2]	{id="296" name="EPS_HAD_BACKUP_Container_ST3" PDUS={ size=1 } }
[3]	{id="297" name="POS_HAD_BACKUP_COM_Container_ST3" PDUS={ size=1 } }
[4]	{id="305" name="ESP_HAD_BACKUP_GW_Container2_ST3" PDUS={ size=3 } }
[5]	{id="312" name="ESP_HAD_BACKUP_GW_Container3_ST3" PDUS={ size=9 } }
[6]	{id="313" name="POS_HAD_BACKUP_GW_Container_ST3" PDUS={ size=2 } }
[7]	{id="314" name="ESP_HAD_BACKUP_GW_Container4_ST3" PDUS={ size=7 } }
[8]	{id="318" name="MPC_R_HAD_BACKUP_COM_Container1_ST3" PDUS={ size=4 } }
[9]	{id="319" name="MPC_R_HAD_BACKUP_COM_Container2_ST3" PDUS={ size=4 } }
[10]	{id="339870104" name="NM_MPC_R_ST3" PDUS={ size=1 } }
[11]	{id="339873791" name="NM_Dummy_ST3" PDUS={ size=1 } }
[12]	{id="373424536" name="ECU_Stat_MPC_R_ST3" PDUS={ size=1 } }
[13]	{id="417437704" name="XCP_Rs_MPC_R_ST3" PDUS={ size=0 } }
[14]	{id="417437705" name="XCP_Rq_MPC_R_ST3" PDUS={ size=0 } }
[15]	{id="417437824" name="DIAG_MPC_R_0E_ExtEth_RS_FD_ST3" PDUS={ size=0 } }
[16]	{id="417437825" name="DIAG_MPC_R_0E_ExtCAN_RS_FD_ST3" PDUS={ size=0 } }
[17]	{id="417437829" name="DIAG_MPC_R_0E_IntEth_RS_FD_ST3" PDUS={ size=0 } }
[18]	{id="417437832" name="DIAG_MPC_R_0E_ExtEth_RQ_FD_ST3" PDUS={ size=0 } }
[19]	{id="417437833" name="DIAG_MPC_R_0E_ExtCAN_RQ_FD_ST3" PDUS={ size=0 } }
[20]	{id="417437837" name="DIAG_MPC_R_0E_IntEth_RQ_FD_ST3" PDUS={ size=0 } }
[21]	{id="417437840" name="Meas_MPC_R1_ST3" PDUS={ size=1 } }
[22]	{id="417437841" name="Meas_MPC_R2_ST3" PDUS={ size=1 } }
[23]	{id="418314376" name="DIAG_RQ_GLOBAL_ExtEth_FD_ST3" PDUS={ size=0 } }
[24]	{id="418314633" name="DIAG_RQ_GLOBAL_ExtCAN_FD_ST3" PDUS={ size=0 } }
[25]	{id="418319757" name="DIAG_RQ_GLOBAL_IntEth_FD_ST3" PDUS={ size=0 } }

Figura 5.1: Class diagram da class Frames

Pode-se verificar que os vetores de ISignals e PDUS estão vazios, pois foram utilizados apenas para preencher o vetor de frames. É possível verificar que este ficheiro ARXML possuía 26 frames, tendo o script conseguido obter o nome, ID e o conjunto de PDUs de cada um (alguns frames não possuíam um conjunto de PDUs). Na figura 5.2, utilizando o frame na posição [5] do vetor (frame identificado pelo nome: "ESP-HAD-BACKUP-GW-CONTAINER3-ST3"), é possível verificar o seu ID, nome e que possui 9 PDUs diferentes.

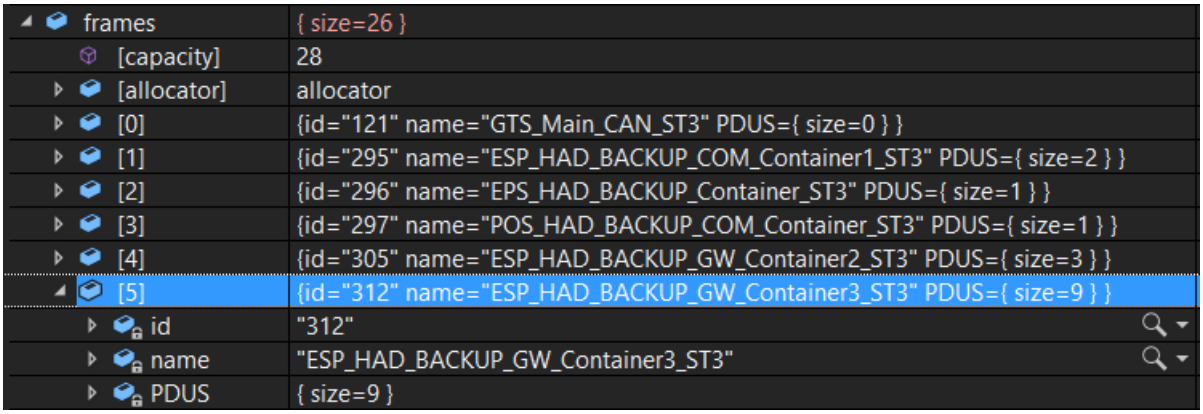


Figura 5.2: Frame ESP-HAD-BACKUP-GW-CONTAINER3-ST3

Na figura 5.3 pode-se verificar o nome dos diferentes PDUS desse frame, tal como o número de ISignals que cada um contém.

Na figura 5.4 pode-se verificar as variáveis do PDU da posição [1] do vetor de PDUs (PDU identificado pelo nome: "ESP-HAD-BACKUP-GW-CONTAINER3-ST3-THC-HVAC-Stat1-ST3"), ou seja, o seu nome e o vetor que contém um conjunto de 15 ISignals.

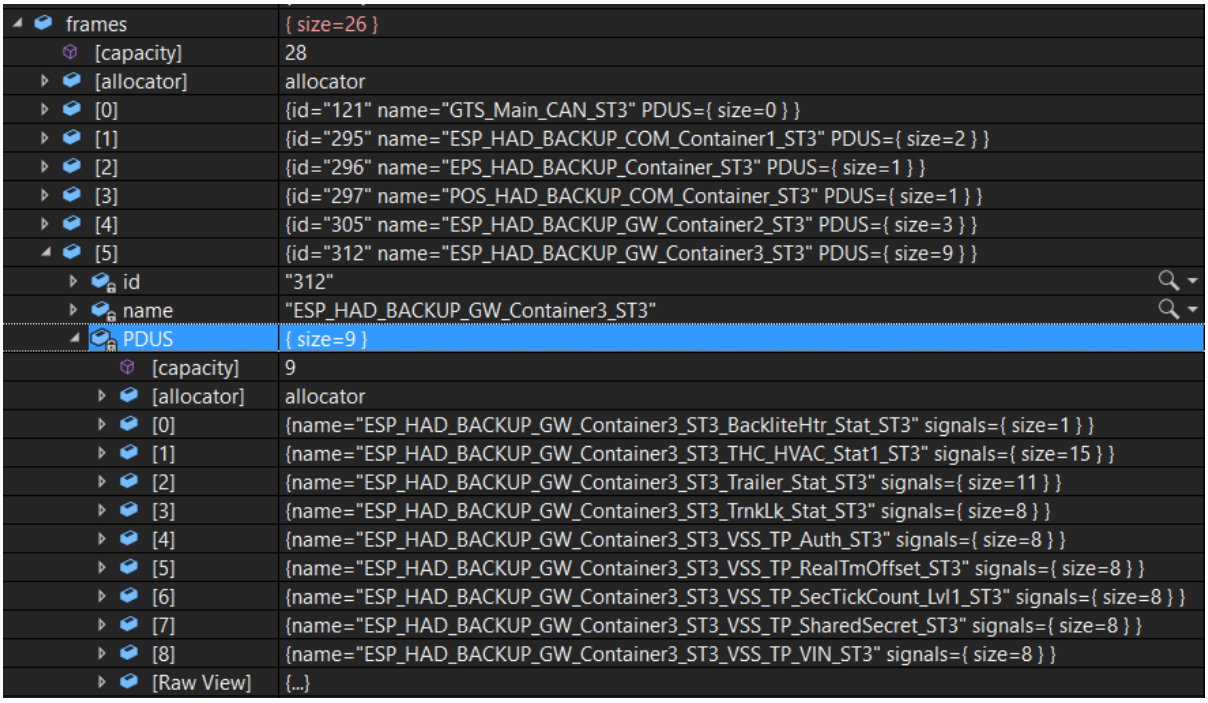


Figura 5.3: PDUS do Frame ESP-HAD-BACKUP-GW-CONTAINER3-ST3

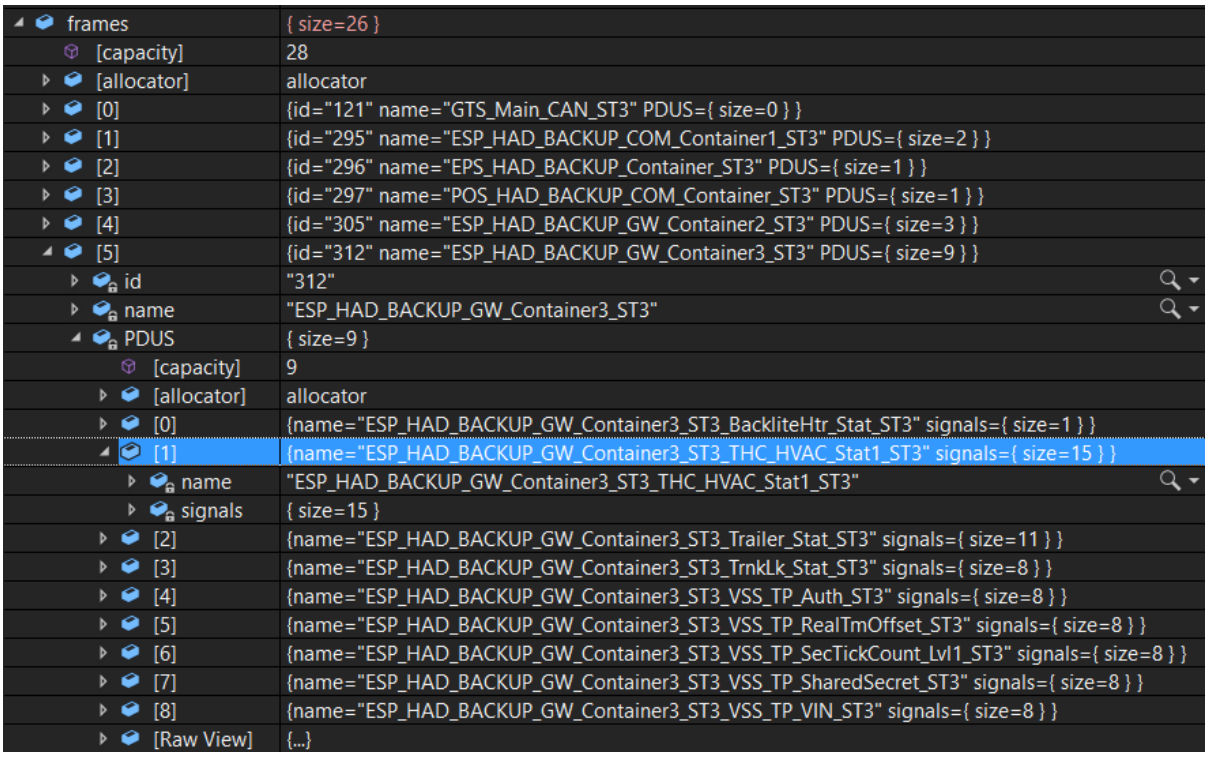


Figura 5.4: PDU ESP-HAD-BACKUP-GW-CONTAINER3-ST3-THC-HVAC-Stat1-ST3

Na figura 5.5 pode-se verificar os 15 diferentes ISignals do PDU em questão.

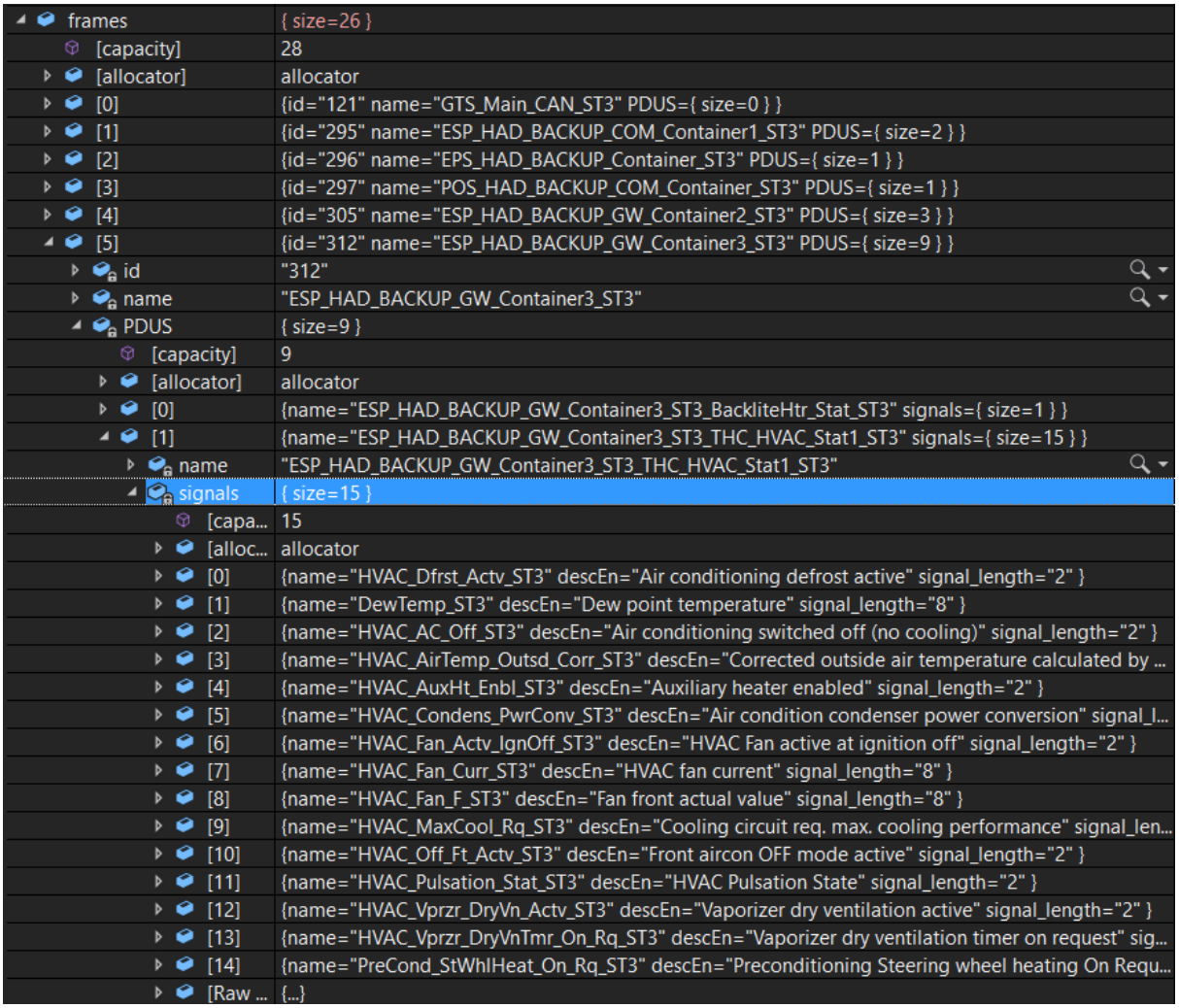


Figura 5.5: 15 ISignals do PDU ESP-HAD-BACKUP-GW-CONTAINER3-ST3-THC-HVAC-Stat1-ST3

Na figura 5.6 pode-se verificar as variáveis do ISignal da posição [0] do vetor de ISignals do PDU em questão. Verificando-se que o seu nome é HVAC-Dfrst-Actv-ST3, com length 2 e com descrição: "Air conditioning defrost active".

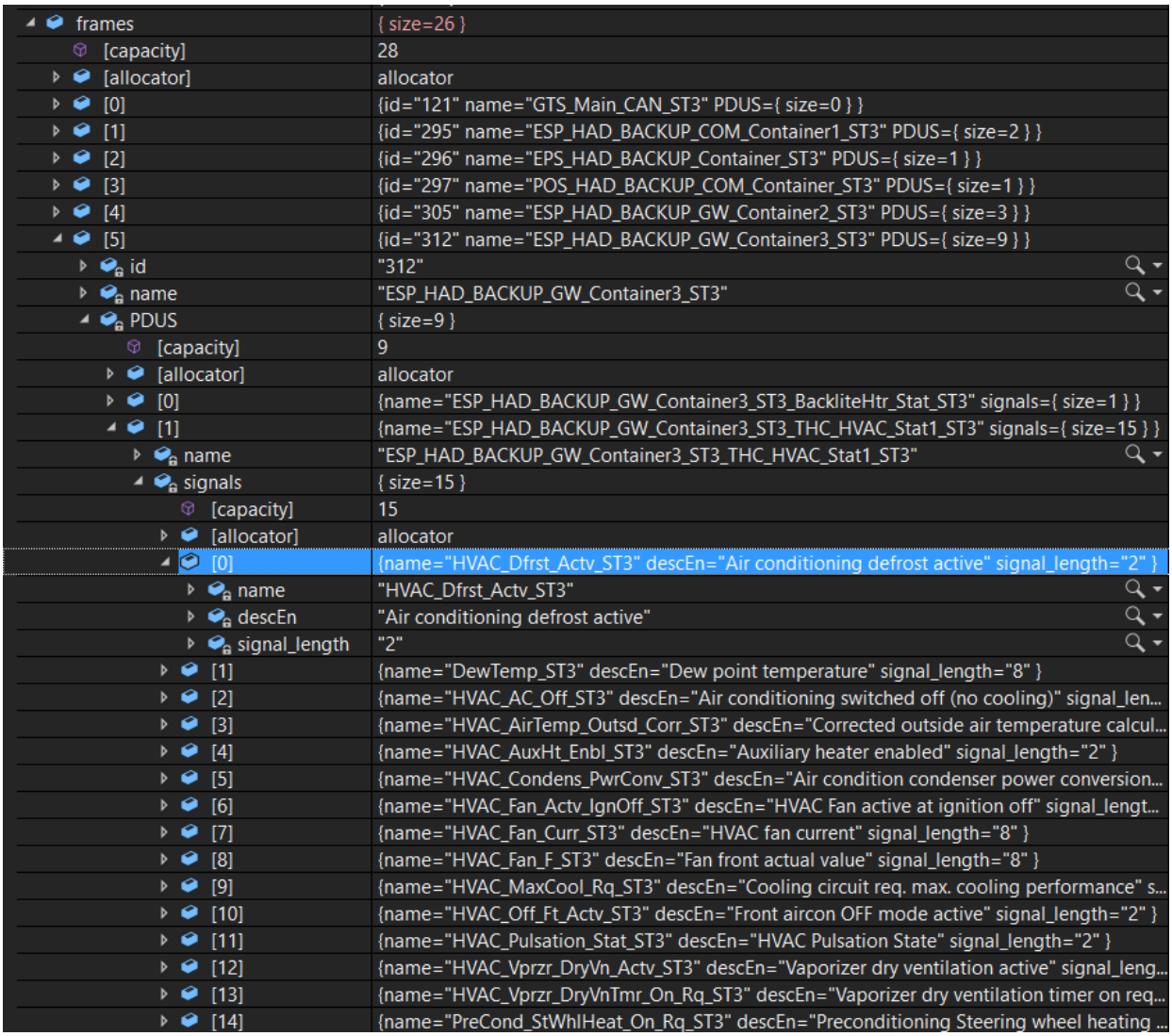


Figura 5.6: ISignal HVAC-Dfrst-Actv-ST3

De modo a se poder validar os resultados obtidos, abriu-se o mesmo ficheiro ARXML utilizado para o desenvolvimento do parser com o software CANoe e pesquisou-se o mesmo frame utilizado como exemplo anteriormente. Comparando a figura 5.7 com os resultados obtidos na figura 5.3, é possível verificar que todos os PDUS do tipo ISignal coincidem com os PDUS que foram parsed pelo programa desenvolvido.

Name	PDU Type	Frame
ESP_HAD_BACKUP_GW_Container3_ST3_BackliteHtr_Stat_ST3	I-Signal-I-PDU	ESP_HAD_BACKUP_GW_Container3_ST3
ESP_HAD_BACKUP_GW_Container3_ST3_ContainerIPdu	Container-I-PDU	ESP_HAD_BACKUP_GW_Container3_ST3
ESP_HAD_BACKUP_GW_Container3_ST3_ESP_HAD_BACKUP_GW_Container3_ST3_Trailer_Stat_ST3_secured	Secured-I-PDU	ESP_HAD_BACKUP_GW_Container3_ST3
ESP_HAD_BACKUP_GW_Container3_ST3_ESP_HAD_BACKUP_GW_Container3_ST3_TrnkLk_Stat_ST3_secured	Secured-I-PDU	ESP_HAD_BACKUP_GW_Container3_ST3
ESP_HAD_BACKUP_GW_Container3_ST3_THC_HVAC_Stat1_ST3	I-Signal-I-PDU	ESP_HAD_BACKUP_GW_Container3_ST3
ESP_HAD_BACKUP_GW_Container3_ST3_Trailer_Stat_ST3	I-Signal-I-PDU	ESP_HAD_BACKUP_GW_Container3_ST3
ESP_HAD_BACKUP_GW_Container3_ST3_TrnkLk_Stat_ST3	I-Signal-I-PDU	ESP_HAD_BACKUP_GW_Container3_ST3
ESP_HAD_BACKUP_GW_Container3_ST3_VSS_TP_Auth_ST3	I-Signal-I-PDU	ESP_HAD_BACKUP_GW_Container3_ST3
ESP_HAD_BACKUP_GW_Container3_ST3_VSS_TP_RealTmOffset_ST3	I-Signal-I-PDU	ESP_HAD_BACKUP_GW_Container3_ST3
ESP_HAD_BACKUP_GW_Container3_ST3_VSS_TP_SecTickCount_Lvl1_ST3	I-Signal-I-PDU	ESP_HAD_BACKUP_GW_Container3_ST3
ESP_HAD_BACKUP_GW_Container3_ST3_VSS_TP_SharedSecret_ST3	I-Signal-I-PDU	ESP_HAD_BACKUP_GW_Container3_ST3
ESP_HAD_BACKUP_GW_Container3_ST3_VSS_TP_VIN_ST3	I-Signal-I-PDU	ESP_HAD_BACKUP_GW_Container3_ST3

Figura 5.7: Frame ESP-HAD-BACKUP-GW-CONTAINER3-ST3 pesquisado no CANoe

5.2 Resultados da Framework

Quantos aos resultados na framework, não foi possível completar a integração do programa com sucesso. O programa foi integrado na core da framework, no entanto não foi possível fazer alterações para adaptar a GUI para ficheiros ARXML, devido a problemas causados por conflitos com o programa desenvolvido. No entanto, as alterações feitas à core da framework na tentativa da integração do programa podem ser evidenciadas. Na figura 5.8 está representado o ambiente de trabalho da framework.

Abrindo a janela para upload da database, representada na figura 5.9, pode-se verificar que a framework permitia o upload de ficheiros ARXML (figura 5.10).

Apesar de não ter sido possível a integração com sucesso do script na framework, é possível especificar como seria a GUI caso este processo tivesse sido executado com sucesso. Na figura 5.11 está representada a GUI para ficheiros DBC.

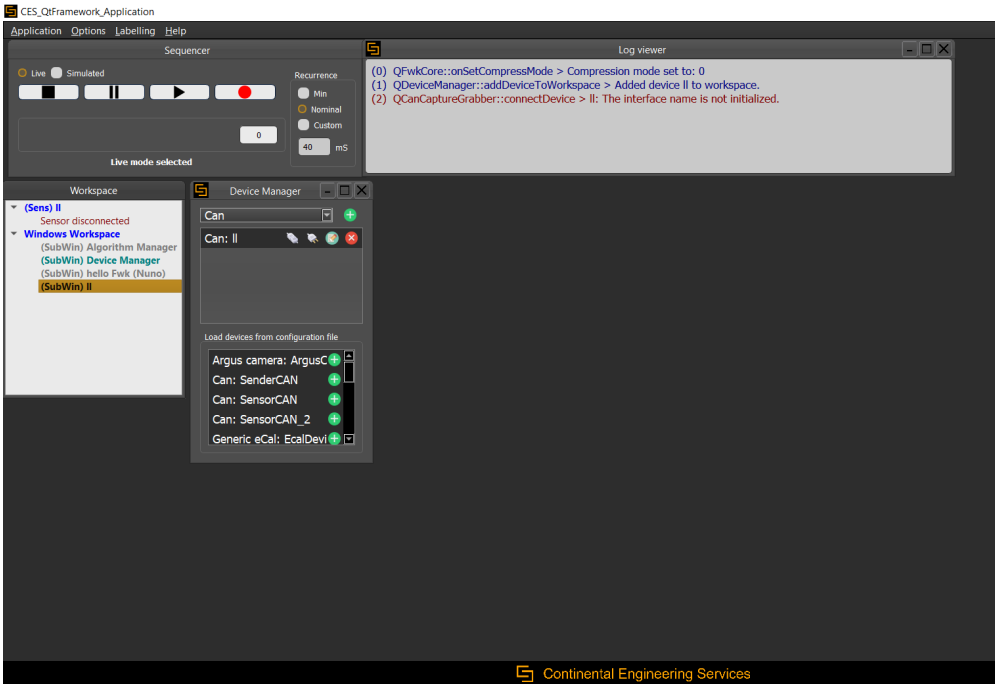


Figura 5.8: Ambiente de Trabalho da Framework

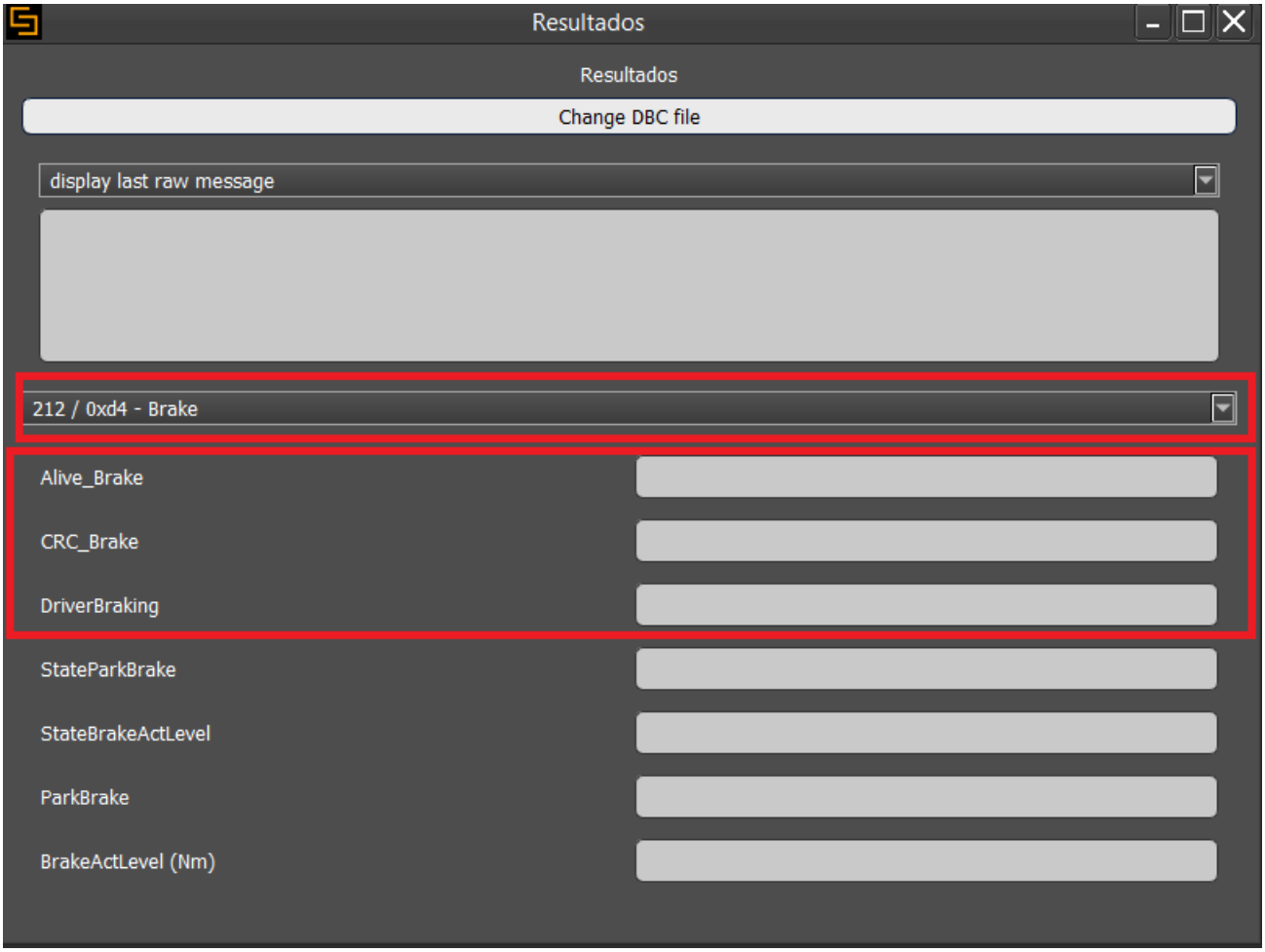


Figura 5.11: GUI para ficheiros DBC

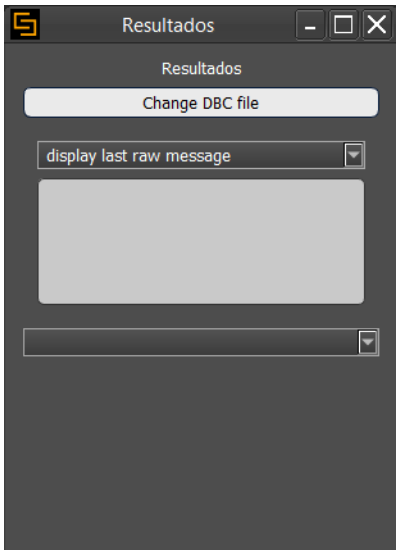


Figura 5.9: Janela para Upload da Database

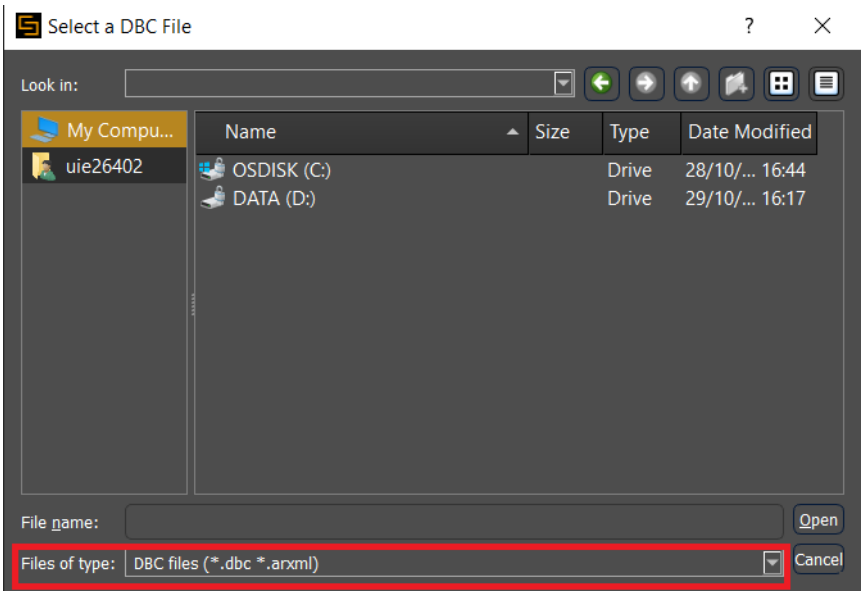


Figura 5.10: Upload de ficheiros DBC e ARXML

Para ficheiros ARXML, a GUI teria uma organização diferente. Para facilitar a explicação, toma-se em conta o frame da figura 5.2, tal como o PDU da figura 5.4 e o ISignal da figura 5.6. Os campos: Alive-Brake, CRC-Brake e DriverBraking seriam o nome, signal length e descrição, ou seja, o nome teria o valor HVAC-Dfrst-Actv-ST3, o signal length seria 2 e a descrição seria "Air conditioning defrost active". O campo 212/0xd4 - Brake seria o ID do Frame e o seu nome, escrito da mesma forma, ou seja, 312 - ESP-HAD-BACKUP-GW-CONTAINER3-ST3,

tal como o PDU que se estivesse a analisar à frente do nome do frame, ou seja, a organização desse campo seria: id - nome do frame - pdu selecionado. Para se navegar entre os frames, clicar-se-ia na setinha à direita do campo da identificação do frame, obtendo uma lista, representada na figura 5.12.

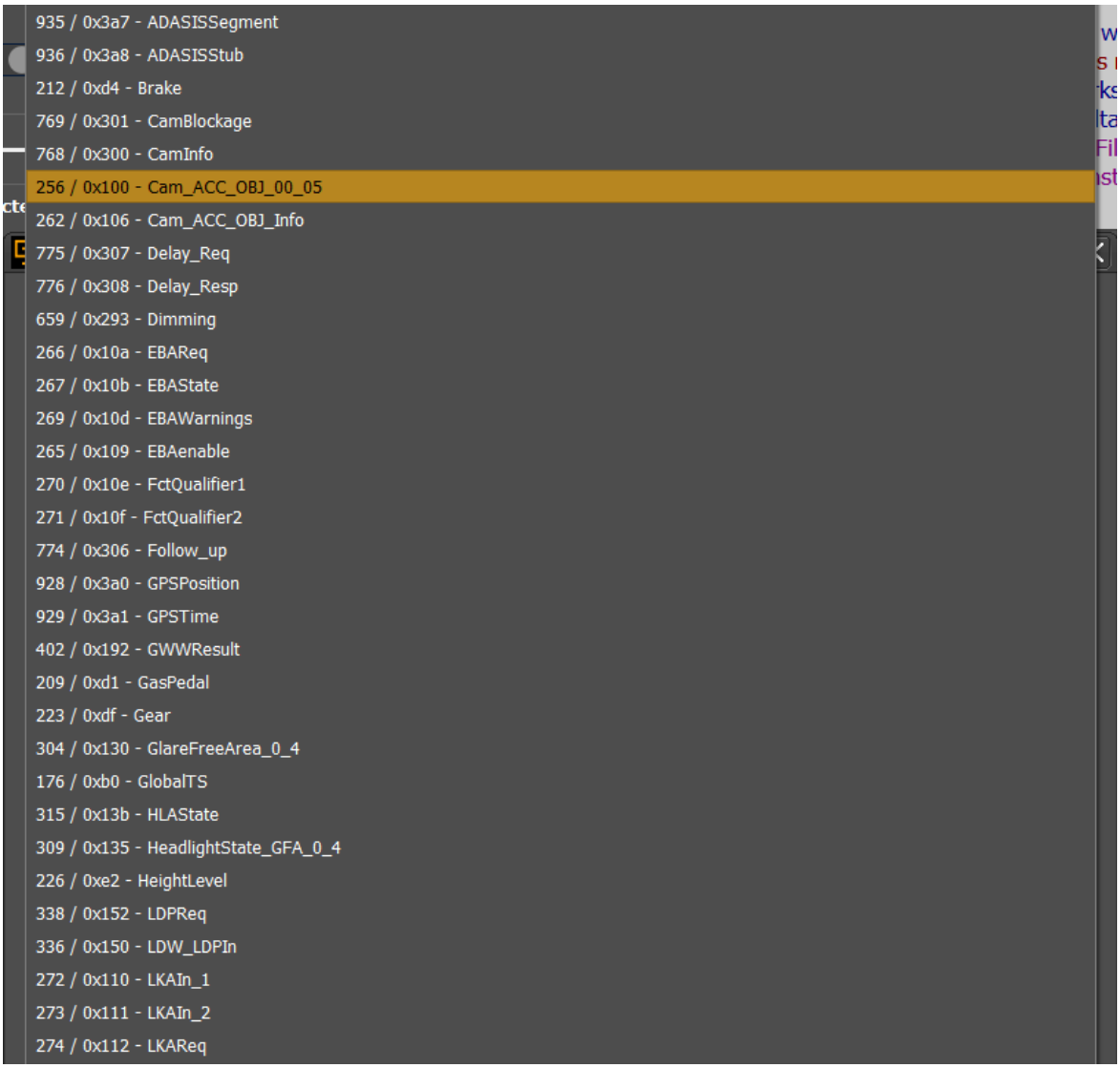


Figura 5.12: GUI para ficheiros DBC (lista de frames)

Nesta lista, seguindo o raciocínio anterior, a organização dos campos seria: id - frame1 - pdu1, id - frame1 - pdu2, id - frame2 - pdu1, e assim sucessivamente.

Capítulo 6

Conclusão

Conforme os objetivos estabelecidos, foi possível fazer o parsing de ficheiros ARXML, obtendo os diferentes valores desejados, no entanto, não foi possível fazer a integração do script na framework com sucesso, devido a problemas causados por conflitos com o programa desenvolvido. Por outro lado, na elaboração do projeto e escrita da tese, foi possível desenvolver competências fundamentais para integrar a equipa de ADAS na Continental Engineering Services, tanto a nível de fundamentação teórica como de programação. O desenvolvimento do programa em C++ permitiu desenvolver aptidões nessa linguagem e a elaboração dos capítulo teóricos permitiu o estudo de conceitos fundamentais como: O que é o AUTOSAR, as Classic e Adaptive platforms, tal como os diferentes protocolos de comunicação: CAN e Ethernet.

Algumas melhorias e trabalho futuro para o projeto são:

- Integração com sucesso do script na framework;
- Suporte de mais versões AUTOSAR;
- Teste do parser ao receber informação em tempo real através do protocolo CAN.

Referências Bibliográficas

[1] Autosar. AUTOSAR History. URL: https://www.autosar.org/about/history/. [cited on p. 4, 5]

[2] Autosar. CLASSIC PLATFORM, 2020. URL: https://www.autosar.org/standards/classic-platform/. [cited on p. 5, 7]

[3] Wesley Chai. Ethernet, 2020. URL: https://www.techtarget.com/searchnetworking/definition/Ethernet. [cited on p. 16]

[4] Continental. Continental Website. URL: https://www.continental.com/en. [cited on p. 1]

[5] Dharmendra Dandotiya. Software Architecture AUTOSAR for Automotive Embedded system share, 2020. URL: https://www.pathpartnertech.com/software-architecture-autosar-for-automotive-embedded-system/. [cited on p. iii, 6, 7, 8, 9, 10]

[6] Doxygen. PugiXml - reading, writing, searching XML data, 2018. URL: https://www.gerald-fahrnholz.eu/sw/DocGenerated/HowToUse/html/group___grp_pugi_xml.html. [cited on p. 20]

[7] Simon Fürst. AUTOSAR the Next Generation – The Adaptive Platform. *PPts*, (September), 2015. [cited on p. iii, 4]

[8] Simon Furst and Markus Bechter. AUTOSAR for Connected and Autonomous Vehicles: The AUTOSAR Adaptive Platform. *Proceedings - 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN-W 2016*, pages 215–217, 2016. doi:10.1109/DSN-W.2016.24. [cited on p. 10]

[9] Brady Gavin. What Is An XML File, 2018. URL: https://www.howtogeek.com/357092/what-is-an-xml-file-and-how-do-i-open-one/. [cited on p. 17]

[10] Elias Hernandez. What Is Adaptive AUTOSAR?, 2020. [cited on p. iii, 11, 12, 13]

[11] File Info. .ARXML File Extension. URL: https://fileinfo.com/extension/arxml. [cited on p. 17]

[12] Marcin Kalicinski. RAPIDXML, 2006. URL: http://rapidxml.sourceforge.net/index.htm. [cited on p. 20, 21]

[13] Arseny Kapoulkine. PugiXML, 2020. URL: https://pugixml.org/. [cited on p. 20]

[14] Lars Knoll. Qt 6.1 Released, 2021. URL: https://www.qt.io/blog/qt-6.1-released. [cited on p. 21]

[15] Ivan Mezei. Cross-platform GUI for educational microcomputer designed in Qt. *Proceedings of 2017 IEEE East-West Design and Test Symposium, EWDTS 2017*, pages 6–9, 2017. doi:10.1109/EWDTS.2017.8110109. [cited on p. 21]

[16] Thomas Mönicke. PQP-QT, 2007. URL: https://www.php-qt.org/. [cited on p. 21]

[17] NI. Controller Area Network (CAN) Overview, 2020. URL: https://www.ni.com/en-us/innovations/white-papers/06/controller-area-network--can--overview.html#section--206487374. [cited on p. iii, 15]

[18] Bijal Parikh. CAN protocol: Understanding the controller area network, 2021. URL: https://www.engineersgarage.com/can-protocol-understanding-the-controller-area-network-protocol/. [cited on p. 14]

[19] Autosar Release. Media Release. *Journal of Christian Education*, os-51(3):67–70, 2021. doi:10.1177/002196570805100309. [cited on p. 5]

[20] Thomas Scharnhorst, Autosar Spokesperson, and Autosar Development Partnership. AUTOSAR for Intelligent Vehicles The AUTOSAR Adaptive Platform. 2018. [cited on p. iii, 10, 11]

[21] Stefan Schmerler and Robert Rimkus. Autosar — Shaping the Future of a Global Standard. *ATZelektronik worldwide*, 8(1):42–45, 2013. doi:10.1365/s38314-013-0147-0. [cited on p. 4]

[22] Continental Engineering Services. Continental Engineering Services Website. URL: https://conti-engineering.com/. [cited on p. 1]

[23] Slant. Best XML parser libraries for C, 2021. URL: `https://www.slant.co/topics/1348/$\sim$best-xml-parser-generator-libraries-for-c`. [cited on p. 19]

[24] Miroslaw Staron. *Automotive software architectures: An introduction*. 2017. `doi:10.1007/978-3-319-58610-6`. [cited on p. 4]

[25] The Apache Software Foundation. Xerces - C++ XML Parser, 2017. URL: `https://xerces.apache.org/xerces-c/`. [cited on p. 19]

[26] Henry Thompson. XML Media Types, 2014. URL: `https://www.rfc-editor.org/rfc/rfc7303.txt`. [cited on p. 18]

[27] Marc Weber. AUTOSAR Learns Ethernet. pages 1–4, 2013. [cited on p. 16, 17]

[28] WxWidgets. WxWidgets - Cross-Platform GUI Library, 2021. URL: `https://www.wxwidgets.org/`. [cited on p. 22]

[29] WxWidgets. WxWidgets - Releases, 2021. URL: `https://github.com/wxWidgets/wxWidgets/releases`. [cited on p. 22]