NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTAMENTO DE ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES

JOÃO FRANCISCO CARVALHO GREGÓRIO

NOVAAS EMBUTIDO

MESTRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES

Universidade NOVA de Lisboa
September, 2023

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTAMENTO DE ENGENHARIA
ELETROTÉCNICA E DE COMPUTADORES

NOVAAS EMBUTIDO

JOÃO FRANCISCO CARVALHO GREGÓRIO

Orientador: Pedro Miguel Negrão Maló
Professor, NOVA University Lisbon

Coorientador: Giovanni Di Orio
UNINOVA

MESTRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES

Universidade NOVA de Lisboa
September, 2023

NOVAAS Embutido

Este documento foi gerado com o processador (pdf/Xe/Lua)LaTeX e o modelo NOVAthesis (v6.10.17) [25].

Agradecimentos

Quero agradecer a toda minha familia e amigos pela força e suporte que me mostraram. Quero especialmente agradecer à minha "Agnes", à minha mãe, ao meu pai e à minha irmã que me apoiaram incondicionalmente durante o processo todo, sem eles nada teria sido possível.

”

«You cannot teach a man anything; you can only help him discover it in himself.»

— Galileo

(Astronomer, physicist and engineer)

Resumo

A Indústria 4.0 é caracterizada pela integração de tecnologias digitais e automação no fabrico, revolucionou o panorama industrial. Esta evolução para a Indústria 4.0 impulsionou a necessidade de sistemas de fabrico mais inteligentes e mais ágeis. Estes sistemas são frequentemente uma implementação de um Asset Administration Shell e são utilizados para monitorizar os sistemas de fabrico. No entanto, as implementações actuais de um AAS carecem de suporte para pequenos dispositivos embutidos porque a maioria deles é construída em linguagens de programação que requerem a presença de um sistema operativo. Esta dissertação estudará os problemas actuais das implementações do AAS e apresentará uma implementação do AAS capaz de ser instalada em dispositivos embutidos. Este AAS compatível com dispositivos incorporados será também testado contra submodelos de referência da indústria para provar a sua eficácia em cenários semelhantes aos da indústria. Esta dissertação também mostra uma aplicação real deste AAS sendo executado num dispositivo embutido para monitorizar em tempo real os níveis de temperatura e humidade de um campo agrícola

Palavras-chave: Industry 4.0, IIOT, Asset Administration Shell, Digital Twin, e-NOVAAS

Abstract

Industry 4.0, characterized by the integration of digital technologies and automation in manufacturing, has revolutionized the industrial landscape. This evolution to Industry 4.0 has driven the need for smarter and more agile manufacturing systems. These systems are often an implementation of an Asset Administration Shell and are used to monitor the manufacturing systems. However, the current implementations of the AAS lack the support for small embedded devices because the majority of them is built on programming languages that require the presence of an Operating System. This dissertation will study the current problems of AAS implementations, and present an AAS implementation capable of being installed on Embedded Devices. This embedded device compatible AAS will also be tested against industry reference submodels to prove its effectiveness in industry-like scenarios. This dissertation also shows a real world application of this AAS being executed in an embedded device to monitor in real time the temperature and humidity levels of an agricultural field.

Keywords: Industry 4.0, IIOT, Asset Administration Shell, Digital Twin, e-NOVAAS

Índice

Índice de Figuras

1

Introdução

1.1 Enquadramento

"A Indústria 4.0 refere-se à ligação em rede inteligente de máquinas e processos para a indústria com a ajuda das tecnologias da informação e da comunicação."(Traduzido de [41]) A nova geração de indústria assenta na crescente tendência para automatizar ainda mais operações de fábricas quando comparado com a geração anterior, e de juntar à crescente automação várias componentes de TI (Tecnologias de Informação).

Indústria 4.0 ou I4.0 incorpora várias tecnologias como por exemplo, AI (Inteligência Artificial) e redes IIoT (Industrial Internet of Things) para recolher informação de plantas de fábricas, culminando num modelo capaz de representar digitalmente uma fábrica. Este modelo é conhecido também como um Digital Twin de uma fábrica.

Para implementar um digital Twin de uma fábrica é necessário que cada elemento da fábrica tenha uma representação digital com comportamento idêntico.

Num esforço de uniformizar e de criar uma arquitectura que sirva de referência para a indústria, foi proposto o RAMI 4.0[32] (Reference Architectural Model for Industrie 4.0). Nesta arquitectura propõe-se que no processo de fabrico, máquinas de fabrico e sistemas de IT trabalhem em conjunto e sejam flexíveis e capazes de customizar produtos mais facilmente. Para aumentar a flexibilidade e customização do produto durante o processo de fabrico, os sistemas de IT vão controlar as máquinas de fabrico, sendo assim necessário que se digitalize estas máquinas para que possam comunicar com os sistemas de IT. À versão digital da máquina que é digitalizada chama-se Digital Twin, conceito que foi referido anteriormente, e para controlar o digital twin, existe o Asset Administration Shell.

O conceito de Asset Administration Shell (AAS), é definido como a representação digital de um ou vários bens[20]. Na estrutura de RAMI 4.0[32] o AAS faz a ponte entre o mundo fisico e o mundo digital, serve também para armazenar a informação relativa ao objeto que está a representar e uniformizar a comunicação do objeto dentro da rede onde está

integrado. Será Vantajoso para definir o digital Twin, que todas as máquinas de uma fábrica sejam representadas e definidas por AAS.

No entanto, entende-se que ao implementar um modelo digital, será necessário que as máquinas sejam compativeis e capazes de comunicar de acordo com o modelo escolhido. Isto significa que talvez máquinas que não foram desenhadas para ter capacidade de integrar uma rede IIoT tenham que ser modificadas para o fazer, ou então mesmo tendo máquinas capazes de integrar uma rede, cada máquina poderá comunicar de forma diferentes, diminuindo a cooperação e coordenação entre máquinas. Estes obstáculos e dificuldades de comunicação são um problema grave porque vão diminuir a eficiência da fábrica.

Obviamente que uma solução aos problemas anteriores é substituir todas as máquinas da fábrica por máquinas que sejam capazes de integrar uma rede e que comuniquem de forma eficiente nessa mesma rede, mas é uma solução inadequada porque vai ter um elevado custo monetário e é um desperdício estar a substituir todas as máquinas apenas para que possam comunicar entre si. Alternativamente, pode-se recorrer a sensores e microdispositivos instalados nas máquinas para modelar os AAS de máquinas e assim uniformizar a comunicação dentro da fábrica. Quando comparado com a solução anterior,esta solução tem um menor custo monetário e não vai criar mais desperdício.

1.2 AAS e os seus constituintes

Para que seja possível integrar um AAS num dispositivo com recursos limitados de tal forma que o dispositivos seja capaz de enviar, receber e interpretar informação, deve-se prestar especial atenção a qual deverá ser a estrutura de dados utilizada. Daí é importante investigar qual será a tipica estrutura de um AAS.

Segundo os Detalhes do Asset Administration Shell[10], o Asset Administration Shell é uma representação digital e estruturada de um objeto. O AAS é o pilar da interoperabilidade entre as aplicações que gerem sistemas de fabrico, porque contém um template da estrutura dos assets que pode gerir, sendo que os assets em si são definidos como instâncias derivadas do template.

Este template é usado como base para vários assets, pode ser composto por várias propriedades que iram definir várias caracteristicas que descrevem e diferenciam assets. Ao conjunto de submodel elements que define um asset e o distingue de outros é considerado um submodel template que poderá servir como referência para outros assets semelhantes a ele.

Para clarificar, vamos basear no exemplo dado em [38] para demonstrar os vários conceitos introduzidos anteriormente. Imagine-se um individuo chamado João Pessoa, suponhamos que se pretende criar um digital twin, podemos criar vários submodelos baseados no individuo, conforme as diferentes situaçoes onde o João Pessoa se pode encontrar. Por exemplo, vamos supor que o João Pessoa pretende ser saudavel, este vai ser um submodel, e as propriedades referentes a saúde do João serão as propriedades do submodel equivalente, propriedades como colesterol, nível de açucar, batimento cardíaco ou pressão arterial. Vamos supor noutra ocasião que o João Pessoa pretende ir de férias a Aveiro, isto irá ser outro submodelo, e vai ser composto por outras propriedades, como por exemplo, o tempo de viagem, meio de transporte escolhido, o alojamento, locais a visitar, a quantidade de malas etc. Mais outro submodelo que compoe o digital twin pode ser a sua situaçao financeira, com mais propriedades como a quantidade de dinheiro no banco, o credito disponivel, as despesas e o ordenado que o João Pessoa recebe. Todos estes três casos estão relacionados com o João Pessoa, de forma análoga, os submodelos estão relacionados ao digital twin do João Pessoa e derivam do mesmo.

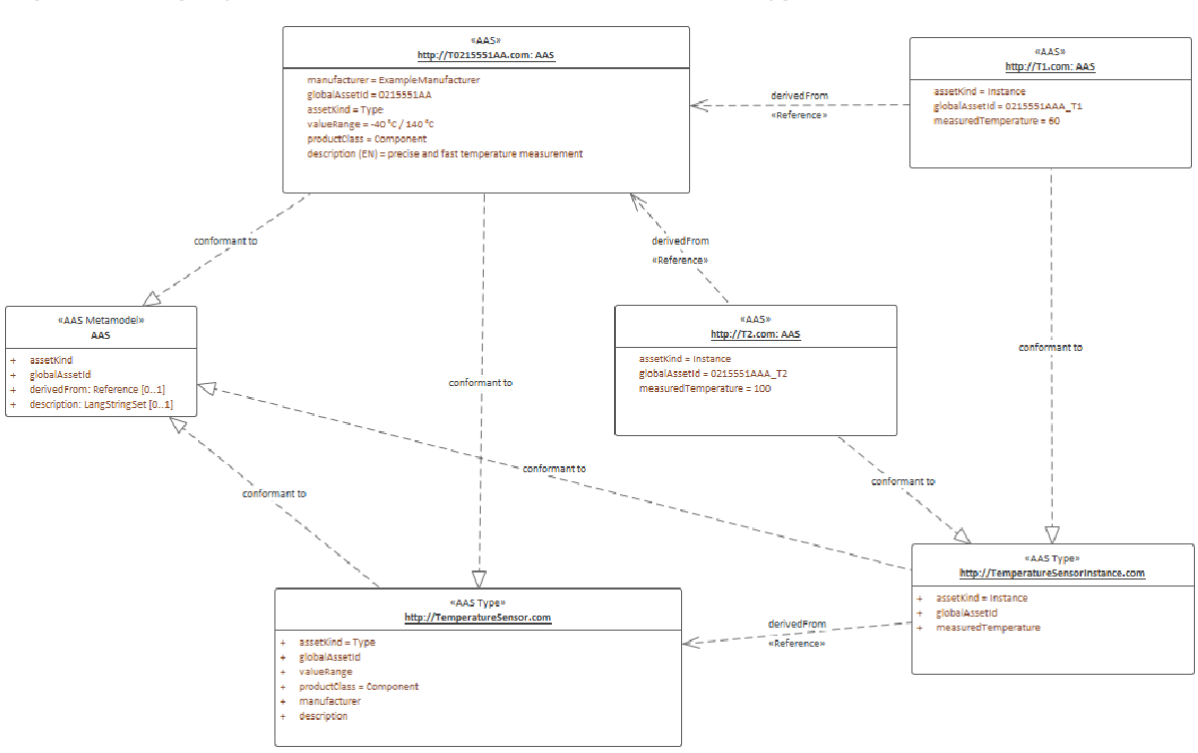

Figura 1.1: Exemplo de como metamodelos se relacionam com tipos de AAS e instâncias de AAS[10]

1.3 Problemas

Como foi referido anteriormente, para implementar digital twins de objetos é necessário que exista um AAS a monitorizar o objeto. Para tal, podemos recorrer a sensores e microdispositivos ligados ao objeto. É importante definir exatamente quais os recursos que vamos utilizar no microdispositivo para ligar os sensores ao objeto a ser digitalizado, é aqui se encontra um problema atual com os programas que gerem e criam AAS, estes programas requerem um sistema operativo para os correr e como tal podem requerer maior quantidade de recursos, memória especialmente, do que é necessário para apenas correr o AAS.

Não existe atualmente um programa que consiga gerar AASs que corram nativamente num microdispositivo sem a utilização de um sistema operativo. Alguns exemplo de programas que gerem e controlam AAS são:

- NOVAAS[28], é um ambiente que implementa e executa AAS, este programa corre em NODE-RED e necessita de um sistema operativo para correr o serviço Node-js que é integral para o NODE-RED funcionar.
- Eclipse Basyx[36][18], desenvolvido pela Fraunhofer IESE, é um middleware SDK desenvolvido principalmente em JAVA, que requer um Sistema Operativo para correr, mas já começa a ser desenvolvido um SDK em C++, o que poderá no futuro permitir o Basyx correr nativamente
- SAP - I40 AAS[31], implementação de AAS feita com imagens Docker de acordo com as referencias de arquitetura estabelecidas no RAMI 4.0, este projeto foi arquivado pela sua comunidade.
- FA^{3}ST[17], O Fraunhofer Advanced Asset Administration Shell (FA^{3}ST) implementa a especificação do Asset Administration Shell definida pela Plattform Industrie 4.0 e suporta modelos AAS customizáveis e fáceis de usar.

A execução de AAS nativamente em dispositivos com recurso limitados traz algumas vantagens. Estes dispositivos têm por norma um custo inferior a computadores ou micro-computadores, por ter um poder de processamento menor também irá consumir menos energia que um computador. A ausência de um sistema operativo também diminui a manutenção do dispositivo porque quaisquer atualizações que possam ser lançadas são apenas feitas pela equipa que desenvolve o AAS. Correr nativamente um AAS também oferece vantagens do ponto de vista da segurança do sistema porque nao existem vulnerabilidades no sistema operativo porque ele não existe.

Assim, devemos clarificar o que definimos como dispositivos de recursos limitados. A Internet Engineering Task Force (IETF)[37], uma entidade de padronização, e que já redigiu

vários RFC (Request for Commments) acerca de dispositivos IoT, define como dispositivos de recursos limitados, dispositivo com limitações Espaço de Memória(Flash/ROM ou RAM), que vai restrigir o tamanho das aplicações que podem ser executadas ou vai restringir a memória disponível durante a execução da aplicação, um dispositivos também pode ser limitado no seu poder de processamente, ou na quantidade de energia disponível.

Visto que os programas de AAS mencionados anteriormente exigem que um sistema operativo esteja presente no dispositivo, nenhum destes programas consegue controlar as classes de dispositivos nativamente. Assim, uma possível implementação terá que muito cuidadosamente conseguir implementar todas as propriedades de AAS e dos objetos de indústria 4.0.

As restrições significativas de recursos, especialmente em termos de memória, podem resultar em problemas substanciais se não forem considerados métodos claros e eficientes na criação de estruturas e objetos deste embedded-NOVAAS (e-NOVAAS).

Devido á possivel complexidade dos objetos de um AAS podem ocorrer alguns problemas devido ao uso excessivo da memória presente no dispositivo ou a uma gestão ineficiente da mesma. Outro problema que pode surgir é o impacto na performance do dispositivo embedded escolhido, assim deve-se tentar minimizar o processamento utilizado durante o momento de execução do AAS. O método usado para criar e alocar espaço para objetos deve considerar o tamanho flexível que os objetos têm sem causar problemas de falta de memória ou degradação do desempenho da placa.

Para além dos problemas anteriores o e-NOVAAS deve também ser compatível com uma grande quantidada de dispositivos, isto significa que não se deve usar bibliotecas ou funcionalidades que funcionem apenas em dispositivos específicos porque elas podem causar conflitos desde perdas de performance ou até mesmo erros no programa que podem impedir o e-NOVAAS de funcionar.

Outro problema que pode ser encontrado está relacionado com o debugging em dispositivos embedded, por vezes problemas de memory leak e outros comportamentos inesperados podem dificultar e prolongar o processo de debugging. Como tal deve ser considerada uma abordagem que simplifique o mais possível o processo de debugging.

Para concluir, implementar um AAS em dispositivos embedded vai apresentar desafios específicos devido ás restrições de recursos presentes. Estas restrições vao exigir que se considere uma abordagem cuidadosa, efficiente e altamente compatível, apenas depois de ultrapassar estes problemas será possível criar um AAS robusto e eficaz.

1.4 Contribuições

A presente dissertação representa um passo em frente no que toca a implementações de AAS em dispositivos embutidos. Este capítulo introdutório destaca as principais contribuições apresentadas ao longo deste trabalho de pesquisa e fornece uma visão geral do impacto que essas contribuições têm no contexto atual da área.

Uma importante contribuição desta dissertação é a apresentação de uma estrutura definida de Submodelos e digital twins para dispositivos embutidos. O estudo destas estruturas pode servir como uma forma de standardizar o processo para desenhar submodelos e digital twins em ambientes com recursos limitados.

A próxima contribuição será a criação de uma ferramenta capaz de criar uma ligação entre a ferramenta de referência para criação de submodelos de digital twins e o ambiente de desenvolvimento embedded. Esta ferramenta é capaz de interpretar qualquer submodelo criado no aasx-package-explorer e criar automaticamente uma estrutura complexa que representa o submodelo no ambiente de dispositivos embutidos, para além desta estrutura também são gerados métodos para inserir e consultar os dados do digital twin.

Uma outra importante contribuição é o piloto demonstrador apresentado no capitulo 5, através deste exemplo podemos entender melhor como funciona um AAS no ambiente de embedded devices.

Ao longo desta dissertação, iremos demonstrar todo o processo que levou á criação destas contribuições, a investigação que foi realizada e também como todas as implementções que foram consideraradas

1.5 Organização da Dissertação

Neste breve capítulo iremos delimitar o que cada um dos capítulos seguintes contêm. Esta dissertação está organizada em capitulos, onde os temas mais relevantes de cada capítulo estão assinalados como subcapítulos.

Para o capítulo 2 é sobre o estado de arte onde vai ser realizada toda a investigação necessária durante o processo de criação da dissertação. Esta investigação inclui um breve estudo sobre como o método de binding escolhido pode influenciar a implementção de uma solução e também inclui um estudo sobre as diferentes implementações de AAS que existem atualmente na indústria.

O capitulo 3 é o capítulo dedicado á implementação de um AAS para dispositivos embutidos que foi desenvolvida. A solução foi chamada e-NOVAAS porque o NOVAAS

serviu como uma forte inspiração. O capítulo contém todo o processo de desenvolvimento e uma explicação detalhada sobre como funciona o e-NOVAAS, contém também outras tentativas de implementação que não foram bem sucedidas, mas serviram para recolher informação importante para a implementaçao final.

O capítulo 4 vai ser dedicado a testes. Aqui o e-NOVAAS será testado para três casos diferentes sendo que cada um irá ser progressivamente mais complexo que o anterior. Estes testes vão demonstrar como o e-NOVAAS consegue ser flexível e fácil de utilizar independentemente da estrutura que se pretende utilizar.

O capitulo 5 demonstra como o e-NOVAAS se enquadra num ambiente real. O e-NOVAAS foi colocado num campo agrícola para monitorizar os dados de Temperatura e Humidade no campo.

O capítulo 6 é o último capítulo da dissertação e vai ser a conclusão da dissertação. Aqui serão consideradas possíveis implementações futuras do e-NOVAAS e algumas funcionalidades que podem ser adicionadas para melhorar o e-NOVAAS. Para além das implementações futuras, vai também ser feito um balanço geral para refletir sobre toda a dissertação.

2

Estado de Arte

2.1 Métodos de Binding

Nesta secção vamos estudar métodos de Binding porque é preciso entender como podemos gerir a memória em dispositivos onde ela é um recurso muito limitado. Mas primeiro, Binding define-se como o processo de associação entre uma entidade e um atributo, ou a associação entre um método e a referência a esse método.[27] O momento em que ocorre o binding, vai determinar o tipo de binding realizado. Ao binding feito durante a compilação do programa, [22]chama-se Early Binding ou Static Binding. Quando o binding é feito durante a execução do programa, chama-se Late Binding ou Dynamic Binding. Iremos analisar estes dois tipos de binding, demonstrando como afetam um programa, com exemplos em C++.

2.1.1 Early Binding

Num processo de Early Binding, o processo de Binding ocorre no momento de compilação do programa, isto é, a associação de métodos ás suas referências é definida antes do momento de execução do programa, enquanto o mesmo está a ser compilado.

Com Early Binding, o compilador vai designar a cada variavel o seu tipo específico de objeto durante a sua declaração criando assim um early-bound object[40], o compilador também vai determinar qual o método que deve ser chamado conforme o tipo do objeto que existe no momento da compilação. Esta atribuição torna o programa mais eficiente e mais rápido na sua execução, esta é principal vantagem de Early Binding.

No entanto, como os métodos ficam definidos antes da execução, isto faz com que os métodos utilizados apenas funcionem segundo a primeira chamada no momento de compilação. Assim, os objetos devem ser adequados á chamada do método utilizado, o que reduz a flexibilidade do programa, ésta é a principal desvantagem de Early Binding.

Uma vantagem de Early Binding é permitir que seja mais fácil detetar erros que possam surgir[4], porque quem desenvolve o programa sabe inicialmente quais os tipos de objetos e funções que vão ser utilizados. Outra vantagem de Early Binding é que permite uma melhor gestão de código porque mais uma vez se sabe quais os tipos de objetos e funções a ser utilizados.

Early Binding[30] pode ter dificuldade a lidar com polimorfismo. Polimorfismo é a habilidade de uma função ser capaz de através de um símbolo de um objeto representar diferentes tipos de objetos[29]. Visto que os simbolos de objetos vão ficar definidos e associados a um tipo de objeto no momento de compilação, é lógico concluir que Early Binding tenha dificuldade a lidar com polimorfismo. Outra desvantagem que Early Binding pode ter, caso o programa faça mais decisões durante o seu funcionamento do que durante a compilação, a performance do programa pode não melhorar ou até mesmo piorar.

Assim, Early binding é um processo útil quando o programador sabe exatamente quais os objetos que devem ser recebidos e utilizados na execução do programa, porque permite melhorar a performance do programa durante o seu funcionamento a custo de dificuldade a lidar com tipos de objetos desconhecidos e flexibiliade nas implementações utilizadas.

Uma técnica muito utilizada em conjunto com Early Binding em linguagens de programação orientadas a Objetos é o method overloading. Method Overloading consiste em criar várias implementações da mesma função com diferentes cabeçalhos, no momento de compilação, o compilador vai identificar as diferentes chamadas ás funções e escolher a chamada com a função que tenha o cabeçalho correspondente[19]. Naturalmente, uma consequência desta técnica e de Early Binding, vai ser a replicação de código de funções para compensar a atribuição que ocorre no momento de compilação, vamos assim ocupar tanto mais espaço de memória quantas vezes forem precisas refazer as funções para aceitar os objetos que o programador sabe que vai receber.

Para Concluir, Early Binding e Method Overloading são noções relacionadas porque ambas involvem escolher a implementação específica de métodos comforme o tipo de objeto ou função. Method Overloading vai permitir que o programador possa ter vários métodos com o mesmo nome, mas cada um com diferentes parâmetros de entrada. Early Binding vai fazer com que o compilador escolha a primeira implementação da função conforme os parâmetros que são utilizados. O programa vai ser incapaz de aceitar objetos que o programador não tenha tido em consideração quando implementou os métodos. O programa é mais rápido na execução mas menos flexível e versátil.

2.1.2 Late Binding

Late Binding[23] é um método de programação onde a implementação de um método chamado por um objeto ou a implementação de uma função chamada por argumentos é verificada no momento de execução do programa. Assim, a chamada fica associada a uma implementação no momento de execução do programa e não é definida durante a compilação como ocorre em Early Binding. Este método é muito utilizado em linguagens de programação dinâmica como por exemplo, JavaScript ou Python. Late Binding também pode ser chamado Dynamic Binding ou Runtime Binding.

Uma vantagem que Late Binding tem é que vai permitir que o programa se adapte a mudanças no tipo ou implementaçao de objetos e funções conforme o contexto da sua chamada. Desta forma, o programa é mais flexível e tem a habilidade de lidar com objetos ou implementações desconhecidas. Late Binding permite assim desenvolver programas mais dinâmicos e interativos porque o tipo ou a implementação de objetos ou funções pode ser alterado ao longo do programa conforme o contexto do momento ou outros fatores externos.

Uma desvantagem que Late Binding vai ter é dificuldade a detetar erros que possam ocorrer durante o funcionamento de funções porque estes erros podem estar a ser causados por tipos de objetos desconhecidos que estão a ser utilizados nas funções, assim pode ser por vezes desenvolver programas com o método Late Binding.[3]

Outra Vantagem de Late Binding é a facilidade de lidar com Polimorfismo, como a associação de um método à sua chamada é feita durante o momento de execução, podemos ter várias implementações de métodos, sendo que segundo o contexto da chamada do método, vai ser escolhida nesse momento a implementação que deve ser utilizada. Esta facilidade torna o programa muito mais flexível comparado a uma abordagem com Early Binding.

A flexibilidade que Late Binding traz, vai infelizmente traduzir-se em pior performance durante a execução do programa. Porque o programa vai fazer as atribuições de referências a métodos e objetos enquanto o programa está a funcionar, vai demorar mais tempo e requer mais poder de processamento.

Late Binding consegue então ser bastante útil porque permite criar programas mais flexíveis e que consigam lidar com tipos de objetos desconhecidos, ao custo de ser mais difícil de resolver certos erros ou bugs e de demorar mais tempo durante a sua execução enquanto são feitas a sua atribuição.

Algumas linguagens de programação orientada a objetos permitem o uso de classes e subclasses. Onde subclasses podem herdar métodos da classe de onde derivam. Por

vezes, subclasses podem ter métodos com o mesmo nome que a classe original, neste caso, será útil que a subclasse seja capaz de substituir os métodos da classe original pelos seus próprios métodos. A esta substituição chama-se Method Overriding.

Com Method Overriding, acontece o seguinte. Quando chamamos um método pela instância da subclasse, vai ser utilizada a implementação do método referente á subclasse, e não a implementação do método que seria normalmente utilizada. Assim, vamos ter métodos com o mesmo nome em várias classes e nas suas subclasses, mas que irão ter implementações diferentes conforme a subclasse que esteja a ser usada no contexto da sua chamada.

Method Overriding é bastante utilizado com Late Binding porque como as associações de métodos a chamadas são feitas no momento da chamada do objeto, podemos adequar a função ao tipo do objeto em específico, até ao nivel da sua subclasse, como sugere [30]. Segundo[9] Method Overriding é usado para dar um comportamento especializado de um método numa subclasse, é uma implementação de polimorfismo nas linguagens de programação orientadas a objetos. Através deste comportamento especializado as subclasses conseguem mudar o comportamento de métodos de acordo com as suas necessidades específicas.

2.1.3 Exemplos e Testes

Vamos agora demonstrar como os dois tipos de Binding vão afetar a forma como classes funcionam, e como afetam a informação utilizada e o funcionamento do programa. Irão ser estudados os métodos em C++, para demonstrar como as duas implementações vão causar diferenças no funcionamento de um programa.

2.1.3.1 Early Binding Exemplificado

Como foi dito antes, em Early Binding, os métodos e objetos são definidos no momento da compilação do programa, assim, [34] em linguagem máquina numa chamada a um método, o compilador vai colocar uma instrução de jump para o endereço desse método. No exemplo seguinte2.1, definimos uma Classe e uma subclasse, Animal e Gato respetivamente. Tanto a classe como a subclasse têm um método chamado display que irá mostrar uma mensagem conforme seja Animal ou Gato. O programa apenas irá pedir que seja criado um objeto Tipo Animal com subclasse Gato e um objeto Tipo Gato.

```cpp
#include<iostream>
using namespace std;
class Animal {
    public:
    void display() {
        cout << "E um animal" << endl;
    }
};
class Gato: public Animal {
    public:
    void display() {
        cout << "E um Gato" << endl;
    }
};
int main(void) {
    Animal *criatura1 = new Gato;
    Gato *criatura2 = new Gato;
    criatura1->display();
    criatura2->display();
    return 0;
}
```

Figura 2.1: Definição do programa a ser compilado e corrido e as suas classes e subclasses

Note-se que C++ é uma linguagem de programação que por defeito utiliza Early Binding. Assim, quando definimos o objeto criatura1, este vai ser declarado com o tipo Animal mas vai ter como subclasse Gato, como no momento de compilação, onde se faz o binding de chamadas a métodos, [24]vai se definir quais os métodos associados conforme o tipo declarado do objeto, independente da subclasse que o objeto tenha. Seguindo esta lógica, é fácil concluir que o método display deste objeto vai então ser o método display da classe Animal e não o que está definido na subclasse Gato, como se verifica na imagem abaixo 2.2.

```
PS C:\Users\ptjfr\CTest> cd "c:\Users\ptjfr\CTest"
PS C:\Users\ptjfr\CTest> & .\"Early_Binding.exe"
E um animal
E um Gato
PS C:\Users\ptjfr\CTest>
```

Figura 2.2: Resultado do programa

Este exemplo também implementa Method Overloading, que é um exemplo de Early Binding em funcionamento. Com Method Overloading, existem vários métodos com o mesmo nome, mas que conforme o tipo do objeto que foi definido durante a compilação

do programa. Daí na imagem 2.2, nota-se que o objeto criatura1 utiliza o método display reservado para objetos tipo Animal e o objeto criatura2 utiliza o método display reservado para objetos tipo Gato, o tipo declarado dos objetos vai definir qual o método que eles podem utilizar.

2.1.3.2 Late Binding Exemplificado

Vamos agora implementar um exemplo semelhante ao anterior, mas utilizando Late Binding. Vai ser implementado também em C++ e para que se utilize Late Binding, é necessário que o método display da classe principal, neste caso a classe Animal, seja definido com a keyword "virtual"[39] porque esta keyword vai certificar que o método é chamada conforme o contexto da sua chamada, não segundo o tipo do objeto que está a tratado. Quando utilizamos a keyword virtual, estamos a utilizar Method Overriding para ignorar o método da classe Animal e utilizar o método da subclasse Gato quando for declarado um objeto que pertença a essa subclasse. Mais uma vez temos dois objeto Animal, criatura1 e criatura2 sendo que o primeiro é também da subclasse Gato. A imagem abaixo 2.3 é o programa que vai ser compilado e executado.

```
#include<iostream>
using namespace std;
class Animal {
   public:
   virtual void display() {
      cout<<"In Animal class" << endl;
   }

};
class Gato: public Animal {
   public:
   void display() {
      cout<<"In Gato class" <<endl;
   }

};
int main() {
   Animal *Creatura1 = new Gato;
   Animal *Creatura2 = new Animal;
   Creatura1->display();
   Creatura2->display();
   return 0;
}
```

Figura 2.3: Definição do programa a ser compilado e corrido e as suas classes e subclasses utilizando Late Binding

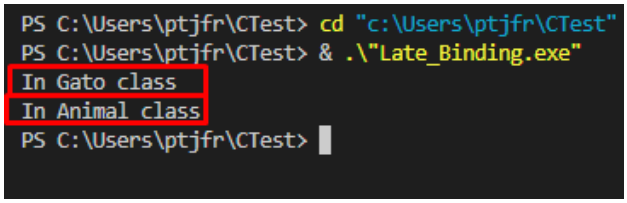

Figura 2.4: Resultado do programa

2.1.3.3 Testes de Velocidade de Compilação e Execução dos diferente tipos de binding

Nesta subsecção iremos analisar os dois métodos de Binding, comparando em C++ os seus tempos de compilação e os tempos de execução, também será analisado o espaço utilizado durante a compilação.

2.1.3.4 Caso a ser Testado

Nestes Testes iremos testar como Early Binding e Late Binding vão afetar o funcionamento de um programa quando se utiliza vários niveis de hierarquia, subclasses de subclasses de subclasses. É importante fazer testes com esta estrutura porque um AAS vai ter vários níveis de hierarquia.

Assim, criou-se uma classe principal chamada Animal, que contém variavéis que correspondem a caracteristicas do animal, aqui chamadas de roll, name, age e marks. Para além destas variáveis, a classe vai ter dois métodos, getter e display, o método getter vai servir para preencher as varíaveis mencionadas anteriormente, o método display irá mostrar no terminal uma mensagem. Note-se que existem também subclasses desta classe, sendo as seguintes, a subclasse Mammal deriva da classe Animal, depois a subclasse Carnivore deriva da subclasse Mammal, por fim a subclasse Canis deriva da subclasse Carnivore. Note-se que cada subclasse tem um conjunto específico dos métodos getter e display, isto serve para mostrar como Method Overloading e Method Overriding e consequentemente os tipos de binding afetam o comportamento do programa.

A estrutura utilizada foi baseada em exemplos vistos em [26] e [6], tendo sido modificada para cada tipo de binding. A imagem seguinte 2.5 é uma representação das classes e subclasses utilizadas

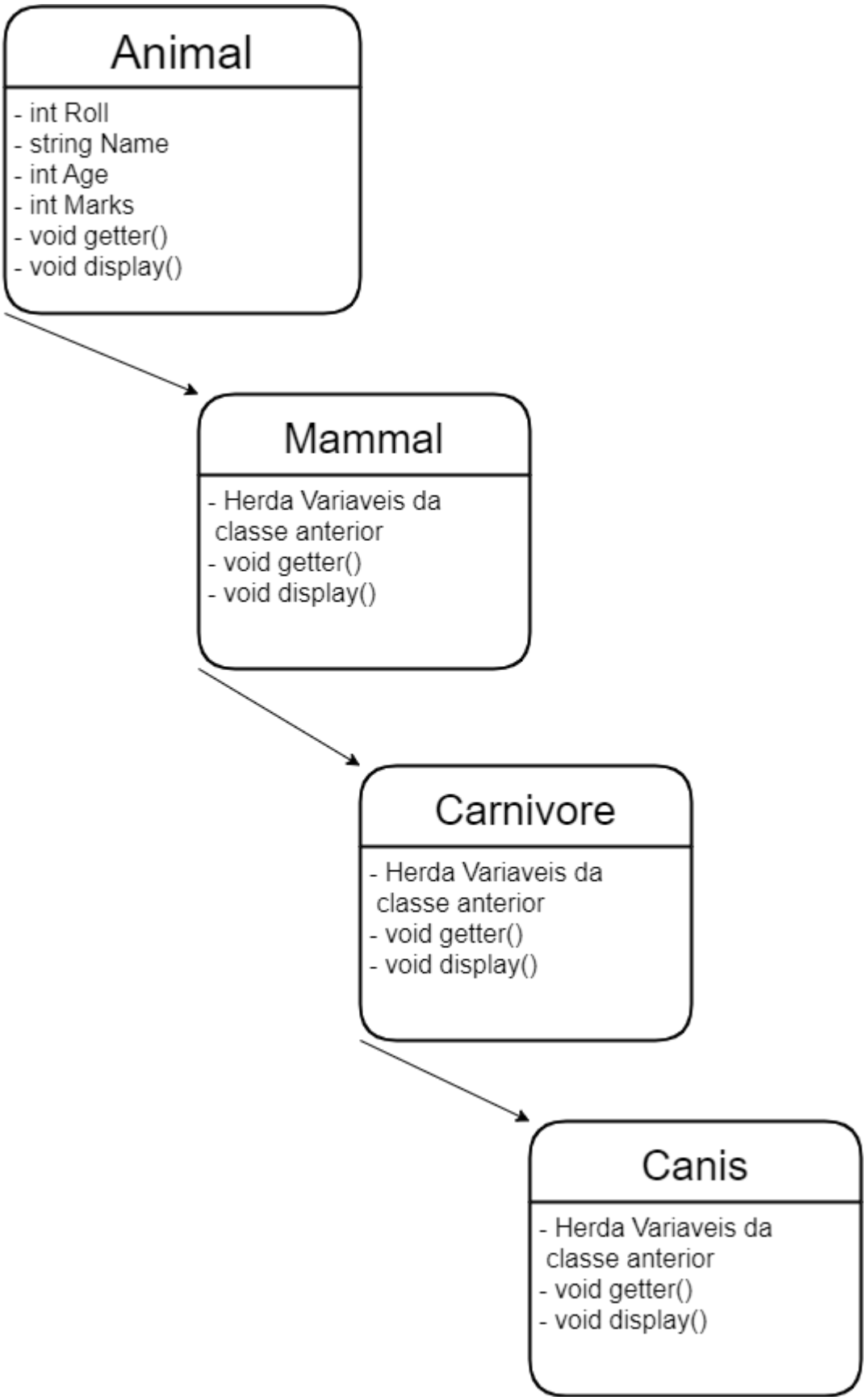

Figura 2.5: Classes e Subclasses

Tendo em conta esta estrutura, o programa vai começar por definir três objetos que pertençam à classe Mammal e as subclasses Carnivore e Canis. Note-se que para o caso do Early binding a classe ou subclasse a que pertençam vai corresponder ao tipo de objeto que lhes é atribuido no momento da declaração como se verifica na atribuição feita na imagem 2.1. A próxima imagem mostra o programa que foi desenvolvido para testar a diferença entre uma implementação Early Binding e Late Binding.

```
int main(void)
{
    clock_t start, end;

    double time_vector = 0;

    for (int j=0;j<100;j++){
        start = clock();
        Animal* criatura1 = new Animal;
        Canis* criatura2= new Canis;
        Carnivore* criatura3 = new Carnivore;
        vector<string> Animal_Vector;
        for (int i=0; i<20000000;i++){

            criatura1->getter();
            criatura2->getter();
            criatura3->getter();

            Animal_Vector.push_back(criatura1->name);
            Animal_Vector.push_back(criatura2->name);
            Animal_Vector.push_back(criatura3->name);

        }
        end = clock();
        Animal_Vector.clear();
        double time_taken = double(end - start);
        time_vector = time_vector + time_taken;

    }
    double avg = time_vector/100;
    cout << "Average Time is:" << avg;
    return 0;
}
```

```
int main(void)
{
    clock_t start, end;


    double time_vector = 0;


    for (int j=0;j<100;j++){
        start = clock();
        Animal* criatura1 = new Animal;
        Animal* criatura2 = new Canis;
        Animal* criatura3 = new Carnivore;
        vector<string> Animal_Vector;
        for (int i=0; i<20000000;i++){

            criatura1->getter();
            criatura2->getter();
            criatura3->getter();

            Animal_Vector.push_back(criatura1->name);
            Animal_Vector.push_back(criatura2->name);
            Animal_Vector.push_back(criatura3->name);

        }
        end = clock();
        Animal_Vector.clear();
        double time_taken = double(end - start);
        time_vector = time_vector + time_taken;

    }
    double avg = time_vector/100;
    cout << "Average Execution Time is:" << avg;
    return 0;
}
```

Figura 2.6: Implementação do programa, Early Binding à esquerda. Late Binding à direita

Após ser feita a declaração das três varíaveis, declaramos também um vetor que vai ser preenchido mais tarde com o name que for atribuido a cada objeto pela função getter. Seguidamente entramos num ciclo for que irá correr 20000000 vezes, sendo que o método getter irá variar de acordo com o objeto usado e é aqui que o binding utilizado irá ser estudado, porque em no caso de Early Binding vamos estar a usar Method Overloading que irá certamente causar uma diferença nos tempos estudados quando comparado com a alternativa do Late Binding. O ciclo acaba colocando a variavel name, obtida em cada criatura com getter, num vetor de Strings, estaremos assim a estudar como o Binding vai afetar o acesso ás varíaveis também. Com isto o ciclo termina. O elevado número de iterações do ciclo foi escolhido para termos uma amostra satisfatória de dados a trabalhar.

2.1.3.5 Método de Teste

Visto que vai ser testado o tempo de compilação e o tempo de execução do programa, é importante saber como vão ser calculados estes tempos. Para calcular tempo de compilação, utilizou-se um comando do compilador GCC chamado -ftime-report[1], este comando ordena o compilador a enviar alguns dados referentes ao processo de compilação, nomeadamente o tempo de execução, o tempo que foi dedicado a chamar métodos e a resolver métodos overloaded e também irá mencionar o espaço em memória utilizado durante a compilação.

Para testar o tempo de execução utilizamos um ciclo que irá correr o programa 100 vezes, combinado com o comando clock()[5], um ao inicio do programa e outro no fim, este comando vai devolver o tempo que foi gasto no programa até ao momento da chamada. Se colocarmos um no inicio e outro fim consegue-se calcular o tempo de execução do programa aos longo das 100 repetições que vai fazer.

2.1.3.6 Resultados

Agora que já foi estabelecido o programa a ser testado e como ele vai ser testado, estamos em condições de testar o programa e analisar os seus resultados.

Comecemos com o tempo de compilação, a imagem seguinte mostra os resultados do comando -ftime-report para Early Binding e Late Binding.

```
PS C:\Users\ptjfr\CTest> gcc -ftime-report -c CTest_Early.cpp

Time variable                                   usr           sys
 phase parsing                        :   1.09 ( 88%)   80475 kB ( 86%)
 phase lang. deferred                 :   0.08 (  7%)    7606 kB (  8%)
 phase opt and generate               :   0.06 (  5%)    5032 kB (  5%)
 |name lookup                         :   0.12 ( 10%)    3260 kB (  3%)
 |overload resolution                 :   0.15 ( 12%)    9900 kB ( 11%)
 callgraph construction               :   0.01 (  1%)    2539 kB (  3%)
 preprocessing                        :   0.34 ( 27%)    2531 kB (  3%)
 parser (global)                      :   0.15 ( 12%)   26426 kB ( 28%)
 parser struct body                   :   0.17 ( 13%)   12811 kB ( 14%)
 parser function body                 :   0.06 (  5%)    4721 kB (  5%)
 parser inl. func. body               :   0.04 (  3%)    2516 kB (  3%)
 parser inl. meth. body               :   0.12 (  9%)    9112 kB ( 10%)
 template instantiation               :   0.30 ( 24%)   29762 kB ( 32%)
 constant expression evaluation       :   0.01 (  1%)     150 kB (  0%)
 integrated RA                        :   0.01 (  1%)    1099 kB (  1%)
 TOTAL                                :   1.24          93997 kB
```

```
PS C:\Users\ptjfr\CTest> gcc -ftime-report -c CTest_Late.cpp

Time variable                                   usr           sys
 phase parsing                        :   0.31 ( 79%)   29780 kB ( 82%)
 phase lang. deferred                 :   0.03 (  7%)    3025 kB (  8%)
 phase opt and generate               :   0.05 ( 12%)    2766 kB (  8%)
 |name lookup                         :   0.05 ( 12%)    1180 kB (  3%)
 |overload resolution                 :   0.03 (  8%)    2209 kB (  6%)
 callgraph construction               :   0.01 (  2%)     233 kB (  1%)
 preprocessing                        :   0.05 ( 13%)    1111 kB (  3%)
 parser (global)                      :   0.06 ( 16%)   11744 kB ( 32%)
 parser struct body                   :   0.06 ( 15%)    4968 kB ( 14%)
 parser function body                 :   0.03 (  7%)    1838 kB (  5%)
 parser inl. func. body               :   0.01 (  3%)     873 kB (  2%)
 parser inl. meth. body               :   0.04 ( 10%)    3100 kB (  9%)
 template instantiation               :   0.09 ( 23%)    9123 kB ( 25%)
 integrated RA                        :   0.01 (  2%)    1111 kB (  3%)
 TOTAL                                :   0.40          36456 kB
```

Figura 2.7: Tempo de compilação do programa, Early Binding à esquerda. Late Binding à direita

Imediatamente, vendo a imagem 2.7 conclui-se que o tempo de compilação para a implementação em Early Binding é bastante superior ao tempo de compilação da implementação Late Binding, dois aspetos que estão a afetar são overload resolution e Template

Instatiation, estes aspetos estão relacionados com a utilização de Method Overloading e de termos objetos do tipo das Subclasses usadas, como está tudo a ser definido neste momento é expectável que demore muito mais tempo do que em Late Binding.

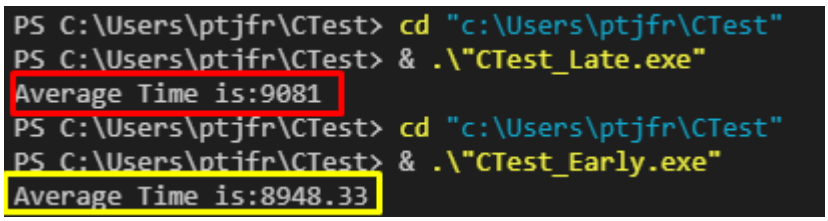

Figura 2.8: Tempo de Execução do programa, Early Binding a amarelo. Late Binding a vermelho

Acerca do Tempo de Execução, as seguintes respostas foram calculadas com o método clock() mencionado anteriormente. Note-se que o método clock() retorna o número de clocks que passaram durante a execução do programa.Observando a figura **??**, notamos que a implementação em Late Binding demora mais tempo que a implementação Early Binding. Uma razão que causou esta demora deverá ser porque em Late Binding os método não estão definidos antes do programa ser executado, assim, o programa tem que verificar qual o método que vai ser usado segundo o contexto atual da chamada, esta verificação quando feita várias vezes vai adicionar algum atraso no tempo de execução do programa.

Para Concluir, estes testes permitiram verificar que a explicação Teórica dada na secção da Explicação Teórica de Early e Late Binding, devemos por fim pensar qual das abordagens poderá ser preferível, será que deve existir um foco para efficiência de tempo de execução ou uma maior flexibilidade a custo de uma diminuição de performance.

2.2 Implementações Existentes de AAS

Atualmente já existem algumas implementações de AAS. Cada uma destas implementações contêm aspetos únicos que as distinguem umas das outras, quer seja pela forma como operam ou pelos dispositivos que suportam. No Capítulo anterior, referimos três implementações distintas. Nesta secção irá ser feito um estudo mais profundo destas três implementações e das funcionalidades que um AAS deve ter.

2.2.1 NOVAAS

O NOVAAS é uma implementação open source do conceito AAS proposto pela RAMI 4.0. Este AAS implementa o conceito do digital twin, pegando num objeto físico e tornando o num componente de industria 4.0 [28]

O NOVAAS foi implementado utilizando Node-Red e quando posto a funcionar pode ser acedido através de uma interface web em qualquer browser. Esta interface contém informações acerca do estado da instância do NOVAAS - se está a comunicar com o digital twin ou houve algum problema na comunicação - e também contém informações acerca do digital twin - as suas propriedades por exemplo.

No repositório do NOVAAS é possível encontrar uma implementação com um conjunto de ficheiros pré-carregados para demonstrar o funcionamento deste AAS. Estes ficheiros depois criam uma interface web que pode ser acedida como localhost para efeitos de teste, caso estivéssemos a utilizar o NOVAAS num produto real, o NOVAAS também pode transmitir informação através de um broker MQTT. A imagem abaixo é o interface Web apresentado pelo NOVAAS para efeitos de teste.

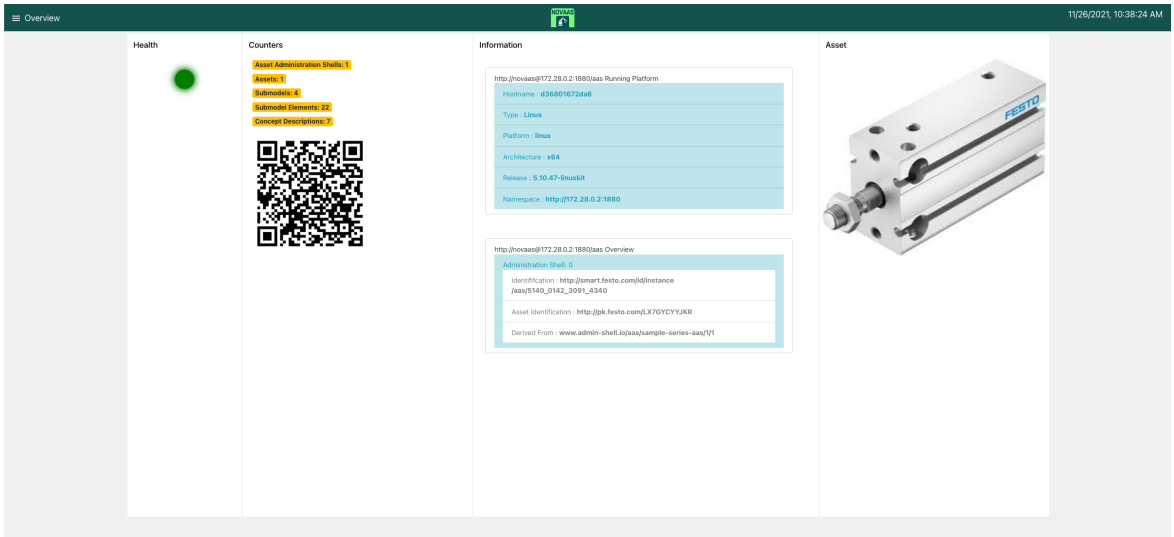

Figura 2.9: Exemplo do Interface Web do NOVAAS[28]

2.2.2 Eclipse Basyx

O Eclipse Basyx é outra implementação open source. Tal como o NOVAAS também implementa conceitos chave como a AAS e digital twins. Segundo a Fraunhofer IESE [36], o objetivo do Eclipse Basyx é ser uma solução que possa ser utilizada por todos os agentes interessados, desde empresas a institutos de pesquisa, ou então universidades e escolas ou até mesmo para apenas individuos que tenham interesse.[18]

O Basyx oferece vários componentes industria 4.0 standardizados prontos a serem utilizados e disponibiliam também um SDK em algumas linguagens de programação, como por exemplo Java, C# e C++, para desenhar componentes personalizados de industria 4.0. Note-se que apesar de existir o SDK para C++, não é explicito em nenhum lado que este SDK pode ser utilizado de forma a que o Basyx possa implementar digital twins e os seus submodelos em dispositivos embedded.

No caso de NOVAAS, era possível comunicar com a interface do AAS através de localhost ou então através de um broker MQTT. No caso do Basyx também é disponibilizado algums meios de comunicação, segundo a página da Eclipse Basyx [18] suporta MQTT, OPC-UA para comunicar diretamente com algumas PLCs que suportem esta comunicação, suportam também comunicação S3 para comunicar com clouds e transportar grandes volumes de dados.[18] Em breve irá também ser suportado PLC4X[18].

Algo que se deve sublinhar acerca do Eclipse Basyx é que se trata de uma implementação muito flexível e fácil de usar porque permite utilizar componentes já pré-construidos o que minimiza o tempo necessário para implementar os mais comuns componentes da indústria, no entanto é também muito flexível porque através dos SDKs disponiveis é possível customizar os componentes que podemos utilzar com o Eclipse Basyx.

2.2.3 SAP - I40 AAS

Outro AAS mencionado foi o SAP - I40 AAS, este AAS foi inicialmente desenvolvido pela SAP mas desde junho de 2022 foi arquivado pela comunidade I4.0 e dividido para dois projetos, uma parte foi para o SDK em Java do Basyx e outra parte foi para o AAS Cloud Repository que é o repositório onde está a implementação que serve como referência para desenvolver AAS. Como foi arquivado e dissolvido para outros projetos, já não é muito utilizado mas continua a ser uma referência importante para o desenvolvimento de AAS e standardização de digital twins e submodelos associados.[31]

2.2.4 FA³ST

Um outra implementação do AAS é o FA³ST[14][17], este AAS foi desenvolvido tal como o Basyx pela Fraunhofer e é caracterizado por ser uma implementação com um excelente suporte para extensões de terceiros. Esta implementação é programada em JAVA e estabelece uma implementação de referência de AAS para esta linguagem.

Esta implementação[16] é caracterizada por ser um conjunto de ferramentas que vão ser utilizada ao longo do tempo de vida de um digital twin. Assim, O FAAAST é composto por vários métodos que iram ser uteis para interpretar e serializar o digital twin a partir do asset inicial. Depois de estruturado dentro do AAS, o FAAAST oferece métodos que suportam vários tipos de comunicação como HTTP e OPC UA.

A imagem abaixo mostra quais são os diferente métodos que o FAAAST utiliza para suportar um digital twin ao longo do seu tempo de vida.

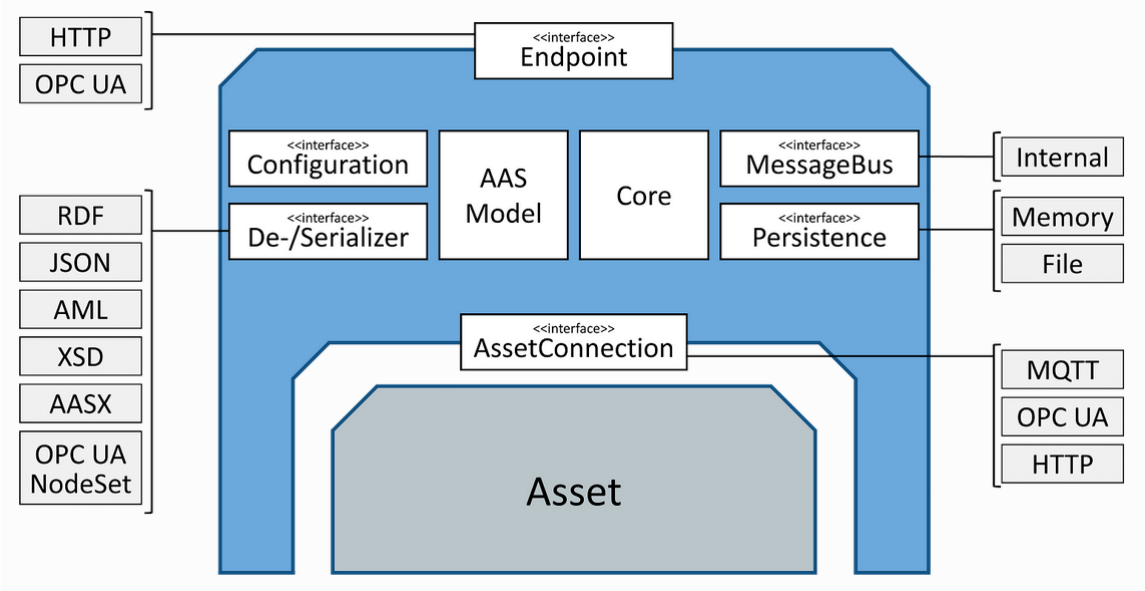

Figura 2.10: Estrutura do FAAAST[12]

Segundo a estrutura [12][13] do FAAAST entende-se que para esta implementação standardiza-se a forma de comunicação ao Asset através dos diferentes métodos de comunicação MQTT, OPC UA e HTTP. Para a comunicação com elementos que estejam acima do AAS, o FAAAST uniformizou a comunicação do endpoint através de um conjunto de métodos que definem uma API segundo a especificação definida no Details of the Asset Administration Shell - Part 2. [35][15]. Esta API vai então permitir que o FAAAST se possa

conectar a qualquer tipo de serviço de uma forma standardizada segundo a especificação da indústria 4.0

O FAAAST também apresenta algumas ideias para possíveis implementações futuras. Para além de melhorias de estabilidade e robustez, também é proposto que o FAAST consiga ser baseado em ficheiros que possam ser guardados em bases de dados para ter uma melhor persitência, outra funcionalidades futuras que também são muito interessantes é a criação de um interface que interaja diretamente com o aasx-package-explorer. [11]

2.2.5 aasx-package-explorer

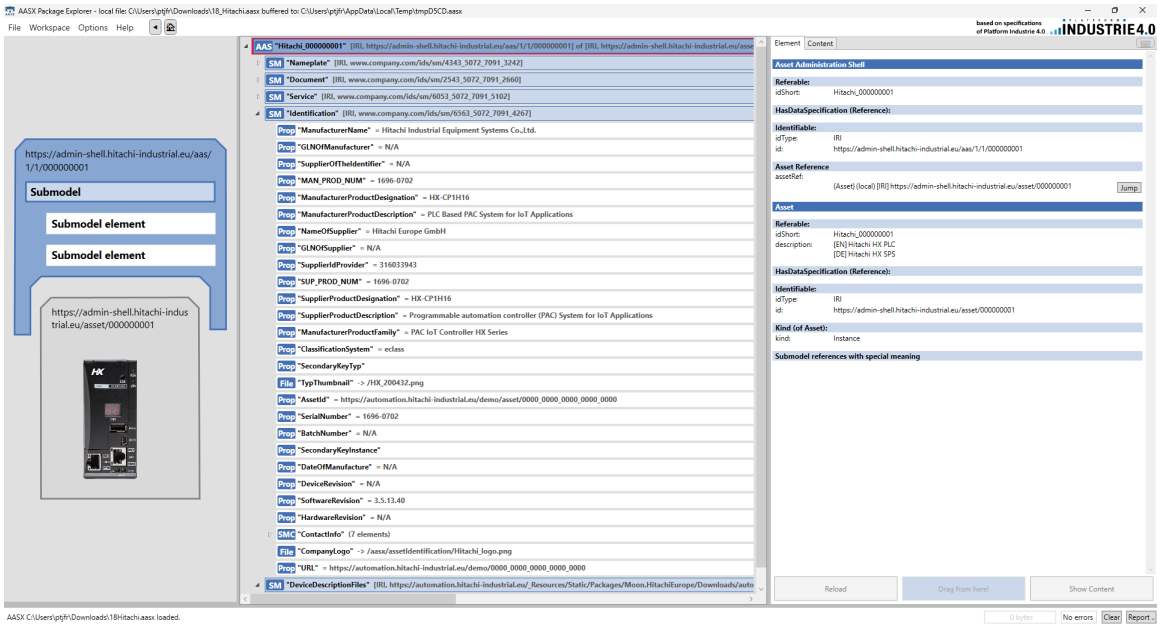

Figura 2.11: Exemplo do UI do aasx-package-explorer

Esta implementação do AAS foi a primeira a definir uma implementação de referência. Foi desenvolvida pela IDTA e é caracterizada por ter uma forte componente gráfica que permite criar, editar e testar Asset Administration Shells segundo as normas estabelecidas para a indústria 4.0.

A componente gráfica do aasx-package-explorer pode ser um bom ponto de partida para gerar estruturas e métodos capazes de serem executados em ambientes embedded porque é capaz de criar um AAS muito rapidamente e ao mesmo tempo respeitando todas as normas existentes. Atualmente, este programa suporta numa versão estável o modelo

V1 dos AAS também suporta em versão alpha o modelo V3 do AAS, porém para esta dissertação iremos considerar apenas AAS que sigam o modelo V1.

2.2.6 Observações e Ideias

Assim, podemos pegar em alguns aspetos destes AAS para servirem como guia e inspiração para desenvolver uma solução que seja tão flexível e com tanta customização como o Eclipse Basyx, que tal como o Basyx seja fácil de implementar os componentes quando eles já existem. Devemos ter em consideração a abordagem definida pelo FAAAST e definir um conjunto de métodos baseados na API definida no Details of the Asset Administration Shell - Part 2. Podemos também considerar a interface Web do NOVAAS como uma possível inspiração para uma possivel implementação de um AAS em dispositivos Embedded.

Uma possível ideia poderia usar o aasx-package-explorer para graficamente projetar um componente customizado mas standardizado de acordo com o modelo estabelecido pela IDTA, assim asseguramos que a solução é flexível. Tendo usado essa ferramenta podemos usar técnicas de geração de código para traduzir a representação gráfica de um digital twin em programação de baixo nível automaticamente, permitindo que a solução seja muito fácil de implementar e por fim, a solução pode ser acedida através de uma simples interface Web muito semelhante á interface do NOVAAS para facilmente reunir a informação útil ao AAS.

3

E-NOVA AAS

3.1 Arquitetura

Nos capitulos anteriores foram sugeridas duas possiveis abordagens, uma onde se gera código e estruturas de acordo com submodelos exportados do aasx package explorer, outra abordagem seria criar uma API conversora capaz de converter dados soltos para uma estrutura de dados compativel com os padrões no RAMI 4.0[32].

Após a análise dos diferentes métodos e abordagens estudadas anteriormente, chegamos à decisão de adotar uma abordagem híbrida que combina características das duas abordagens, no entanto foi dado um maior enfâse no desenvolvimento de um potente gerador de codigo. Assim, a abordagem proposta será um gerador de codigo de estruturas e métodos que durante o periodo de execução do programa e conforme a necessidade do utilizador cria novos objetos.

Esta abordagem hibrida resulta num programa capaz de interpretar submodelos provenientes da ferramenta aasx-package-explorer e gerar as estruturas de código e os métodos que representem um submodelo na linguagem de programação que escolhermos para executar o AAS. Inicialmente, considerou-se gerar estruturas em Rust e C++, mais tarde provou-se que C++ seria uma melhor escolha, este tópico explicado mais tarde quando se relatar o processo de desenvolvimento em cada linguagem.

O código gerado vai ser utilizado para duas funções, as estruturas de código são uma representação dos submodelos escolhidos e servirão como o motor para processamento de informação enviada por uma máquina que não comunique de acordo com os padrões de um AAS, por outro lado os métodos gerados que serão usados pela estrutura servem como uma API.

Indenpendente da linguagem de programação escolhida algo que se deve compreender será como construir uma API que siga padrões ja estabelecidos pelas indústria. Segundo o 2º documento publicado pela Industrial Digital Twin Association acerca das especificações

do Asset Administration Shell[35], ficou definida uma API ou o protótipo de uma API para AAS e submodelos. Esta API vai servir como guia de como interagir com os principais métodos gerados.

A definição da API[2] feita pela IDTA contém várias funções importantes para monitorizar um submodelo. Para efeitos de prova de conceito, o gerador implementa apenas os métodos básicos de criar, ler e apagar submodelos, o método para alterar ou atualizar campos de um submodelo pode ser feito como se alterássemos o valor de uma variável normal em programação. Note-se que alterar o submodelo não tem grande utilidade visto que o AAS é mais utilizado para obter e enviar dados e não afetar, no entanto existe a possibilidade de afetar os campos das estruturas dos submodelos caso seja desejado.

Desta forma, a imagem abaixo serve como legenda para a arquitetura do embedded NOVAAS.

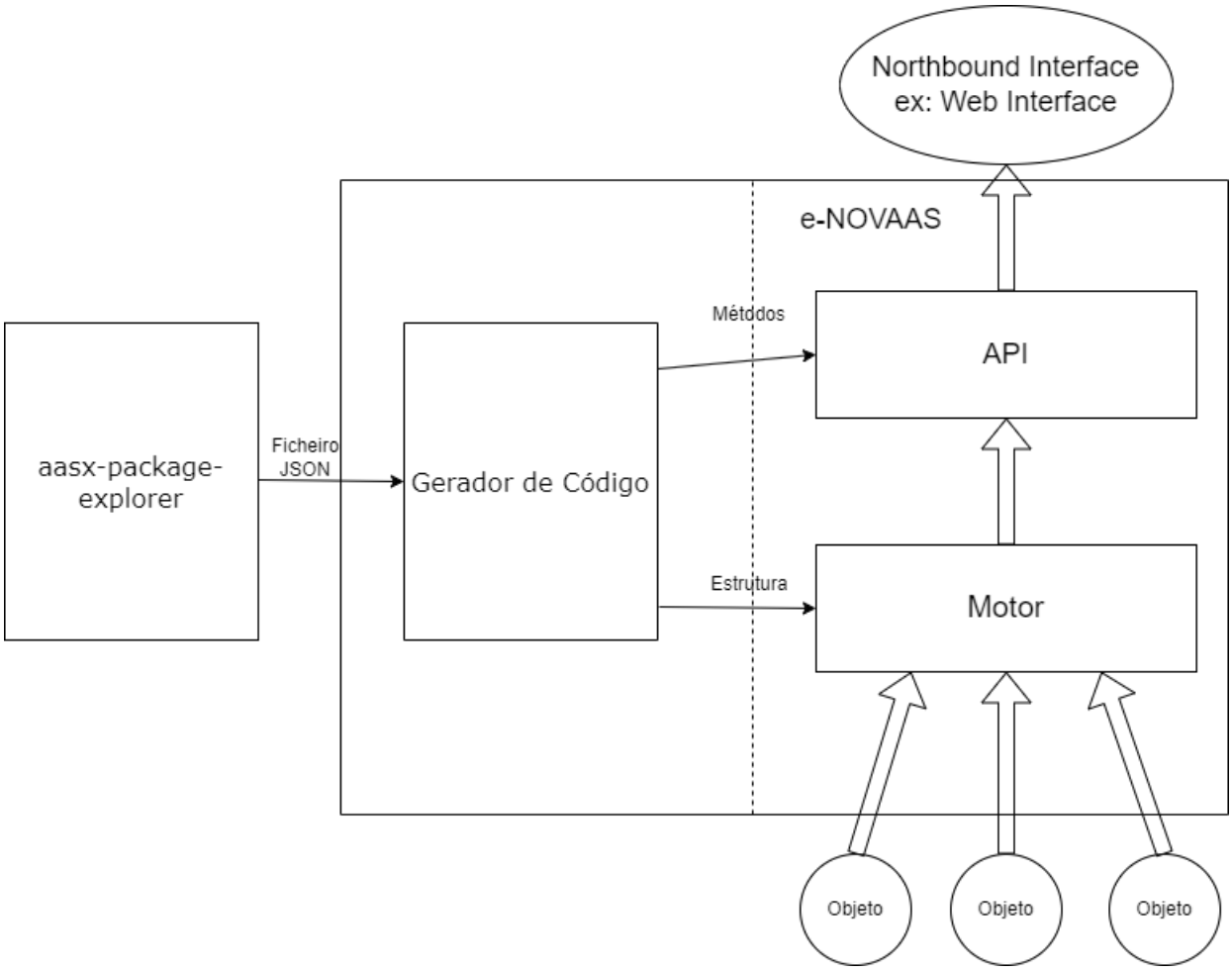

Figura 3.1: Arquitetura de funcionamento do e-NOVAAS

3.2 Implementação

3.2.1 Introdução

Tendo delineado a arquitetura do e-NOVAAS vamos entrar em maior detalhe da implementação desenvolvida. Para além disso, iremos detalhar o processo de extrair submodelos do aasx-package-explorer, como utilizar o gerador de código e o e-NOVAAS e quais foram os desafios e dificuldades encontradas tal como as linguagem de programação usada no gerador e do código gerado.

O objetivo da implementação é demonstrar que é possível a implementação de estruturas e métodos que sejam uma representação idêntica de submodelos de digital twins. Para tal, será necessário compreender a estrutura que um submodelo tipico tem. Anteriormente referiu-se que submodelos podem derivar de digital twins, sendo diferentes representações de digital twins, estes submodelos podem ser compostos por elementos como por exemplo, propriedades, coleções de elementos de submodelos, eventos.

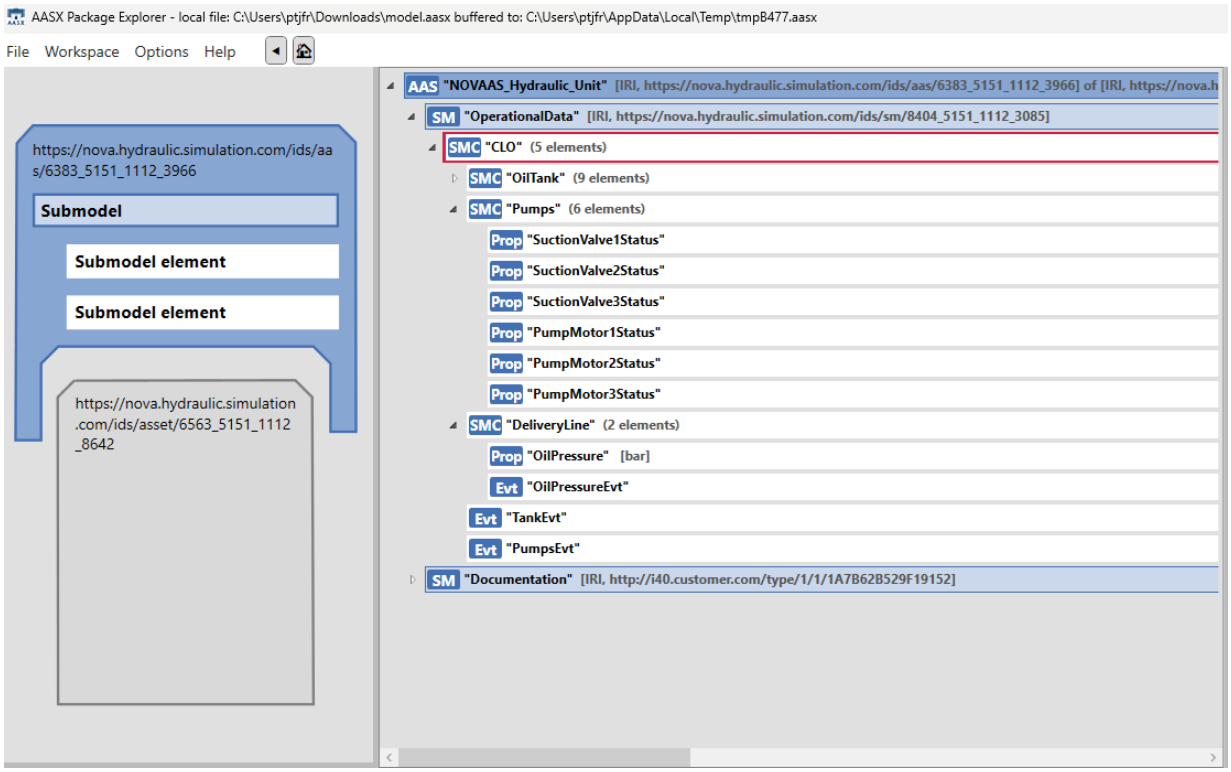

Figura 3.2: Submodelos, Propriedades, Coleções de elementos de submodelos e Eventos representados no aasx-package-explorer

Cada elemento de submodelos tem por norma um comportamento diferente do outro. Como tal, é necessário compreender quais os mais importante para desenvolver uma

prova de conceito e qual seria a sua melhor representação em código. Propriedades por exemplo tal como o nome indica são propriedades correspondentes a um submodelo ou a uma coleção de elementos, desta forma numa estrutura as propriedades representam um membro de uma estrutura. Uma coleção de elementos de submodelos é tal como o nome indica um subgrupo de elementos de submodelos podem ser por exemplo, constituidos por propriedades ou eventos, ou então até mesmo outras coleção de elementos de submodelos, sendo assim a representação em código de uma coleção de elementos de submodelos seria uma "nested structure"uma estrutura dentro da estrutura principal.

Agora que estabelecemos alguns paralelismos entre conceitos dos AAS e código vamos apresentar como se pode extrair um submodelo do aasx-package-explorer e usá-lo para gerar código.

3.2.2 Exportação de Submodelos

É muito fácil extrair submodelos do aasx-package-explorer, a ferramenta da IDTA tem a funcionalidade de exportar submodelos no formato JSON. Para tal, devemos selecionar o submodelo desejado, depois usamos a tab "file", onde encontramos a função "Export.."e a subcategoria "Export Submodel to JSON". Assim o submodelo vai ser exportado em formato JSON para uma pasta escolhida pelo utilizador, preferencialmente deve ser a mesma pasta onde está o gerador de código.

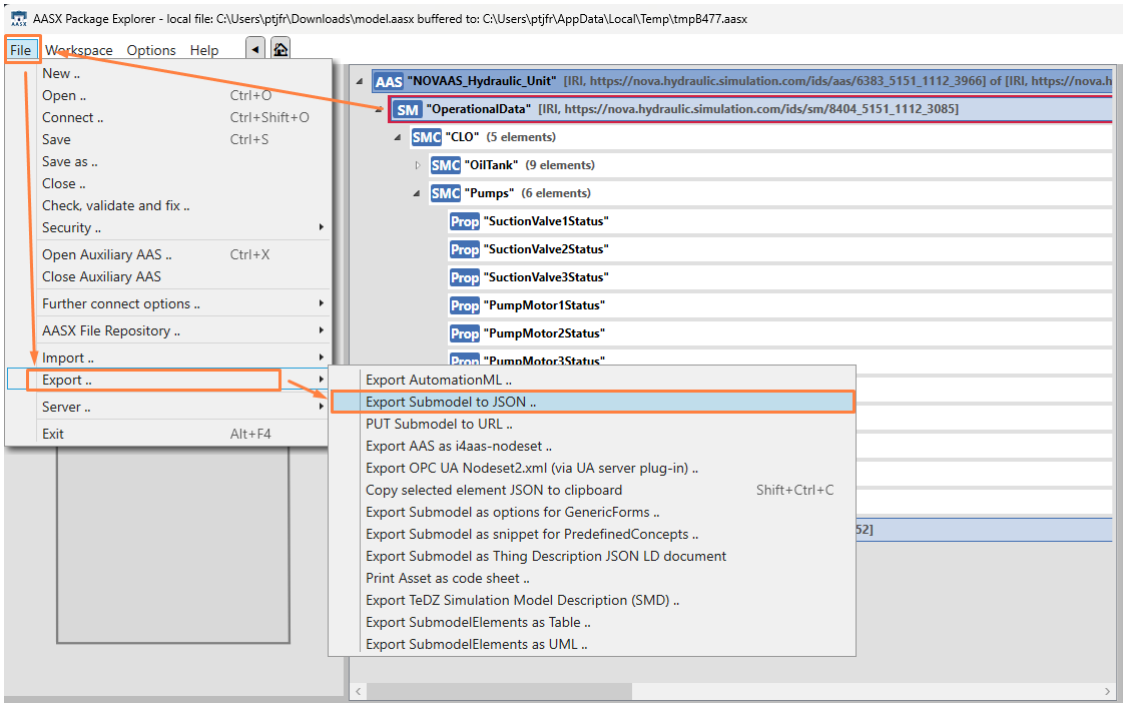

Figura 3.3: Processo de exportação de submodelos do aasx-package-explorer

Algo que devemos notar acerca do aasx-package-explorer é que os submodelos criados são criados segundo a versão V3.0RC01[7] como tal o gerador de código apenas irá funcionar com submodelos que sejam criados segundo esta mesma versão.

3.2.3 Gerador de Código

O gerador de código é o ponto principal do e-NOVAAS, ele é utilizado para gerar estruturas de código tendo como base submodelos exportados do aasx-package-explorer. Vamos começar por analisar o submodelo exportado, este submodelo vai ser analisado pelo gerador de código e de acordo com vários campos do mesmo, abaixo está a estrutura em json do submodelo exportado mostrado na figura 3.3:

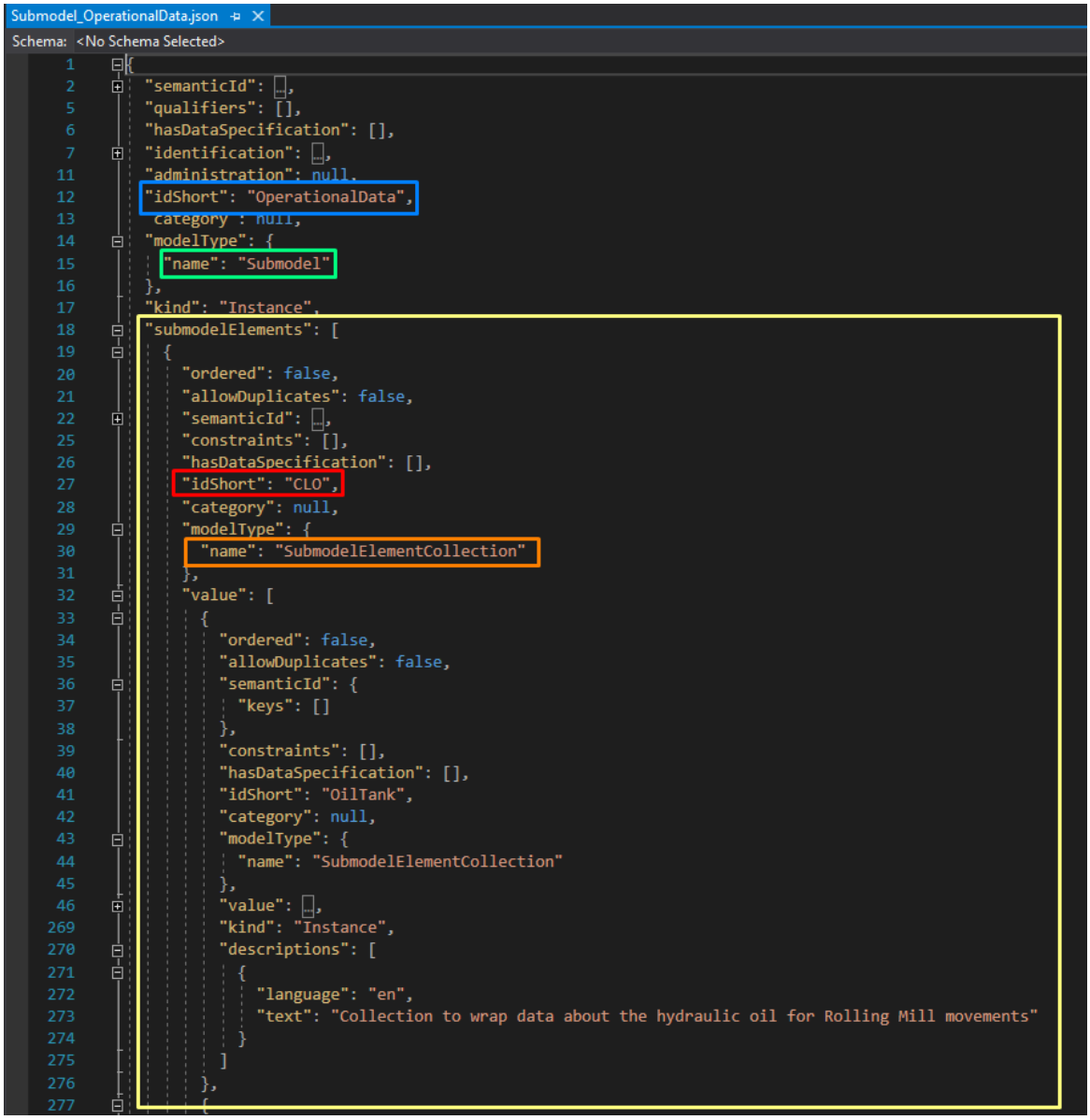

Figura 3.4: Submodelo em formato JSON

Analisando a figura 3.4 temos delineado com retangulos coloridos os diferentes aspetos interessantes para o gerador de código criar estruturas e métodos.

Assim, temos a azul o campo "idShort"que o gerador usa como o nome da estrutura principal por este ser o nome do submodelo, sabe-se que se trata do submodelo porque a verde temos o subcampo "name"do campo "modelType"a identificar que se tratar de um submodelo.

Os elementos do submodelo estão sempre no campo "submodelElements"delineado com um retângulo a amarelo. Este submodelo apenas tem um elemento presente, se houvessem mais elementos cada um estaria inserido numa lista separados por vírgulas.

Dentro de cada elemento temos mais uma vez o campo "idShort"que vai identificar o elemento, o tipo do elemento é identificado pelo campo delineado a laranja "name". Como é um SubmodelElementCollection isto significa que se trata de uma coleção de elementos de submodelo, imediatamente abaixo do campo anterior temos o campo "value"que para o caso de uma coleção de elementos seram os elementos que compõem a coleção. Em código esta coleçao seria um campo na estrutura inicial com o nome do submodelo que apontaria para uma estrutura com o nome da coleção, neste caso "CLO"composta pelos elementos visto no campo "value"da coleçao, neste caso um dos elementos é "OilTank", caso existam mais elementos na coleção o método de identificação é igual, sendo que cada vez que encontramos um novo SubmodelElementCollection criamos uma nova estrutura com o nome da coleção encontrada.

Assim, o funcionamento do gerador de código é o seguinte: o programa lê o ficheiro que contém o submodelo em formato de texto (.txt) procurando inicialmente o submodelo e os constituintes do submodelo, criando uma estrutura com o nome do submodelo e cria também os métodos para inserir e ler objetos e dados da estrutura, métodos INSERT e GET. Depois, os elementos do submodelos são analisados e caso sejam coleções de elementos é criada uma estrutura nova com o nome da coleção e o processo repete-se as vezes necessárias até chegarmos a um nivel do submodelo onde apenas existam propriedades e não mais coleções.

Quando o gerador encontra uma propriedade ele atribui à mesma um campo na estrutura e a propriedade vai ser um tipo comforme o tipo de dados apresentado no campo "name"do subcampo "dataObjectType"do campo "valueType"da propriedade, para strings por exemplo iremos atribuir o equivalente às mesmas na linguagem de programação escolhida, para valores inteiros atribui o tipo de inteiro, por defeito ou se o tipo nao for identificado a varíavel fica com o tipo equivalente a um float, a imagem abaixo mostra os campos mencionados de uma propriedade.

Figura 3.5: Exemplo de uma propriedade de um submodelo em formato JSON

Observando a figura 3.5 encontramos os campos mencionados anteriormente, a Laranja verificamos que esta propriedade pertence à uma coleção de subelementos "OilTank", a vermelho verificamos que a propriedade é "OilLevel"e a azul os campos referentes ao tipo de dados que a propriedade é, como não está especificado esta propriedade será um float.

Para cada estrutura criada são paralelamente gerados métodos INSERT e GET especificamente desenhados para essas mesmas estruturas, note-se que podem existir métodos que dependam de métodos de outra estruturas dependentes, nesses casos o gerador irá chamar os métodos correspondentes a essas mesmas dependências no método original.

Deve-se notar uma particularidade acerca dos métodos INSERT, durante o desenvolvimento foi decidido que os métodos INSERT recebem uma string que contém informação referente ao submodelo carregado no e-NOVAAS no formato JSON, ou seja, segundo a figura 3.2 os objetos enviam informação para o motor no formato JSON e os métodos são capazes de inserir a informação na estrutura do submodelos automaticamente. No entanto, caso o utilizador prefira ou não seja capaz de enviar informação em formato JSON é também possivel inserir manualmente os dados nas estruturas.

Outro ponto importante acerca destas mensagens JSON é que dependendo da complexidade dos objetos a mensagem recebida pode ser muito comprida e o utilizador pode não querer preencher a estrutura inteira ou então pode querer usar o método INSERT para atualizar um campo em especifico de uma parte de uma estrutura, isto é possível fazer porque o gerador de código gera funções capazes de interpretar estas mensagens parciais.

Tendo mencionado agora como funciona o gerador de código e como o mesmo interpreta submodelos, iremos mostrar as duas versões do gerador de código.

3.2.3.1 Gerador de código Rust

Esta primeira implementação do gerador de código, é capaz de gerar estruturas de acordo com o submodelo dado, mas não gera métodos de INSERT ou GET. As estruturas geradas utilizam uma biblioteca no-std do rust chamada serde-json[33] para deserializar (Converter json string para objeto da estrutura) ou serializar (Converter objeto da estrutura numa json string) variáveis.

O gerador tem ainda a capacidade de preencher apenas preencher certos campos da estrutura caso exista a mesma necessidade. Esta funcionalidade é permitida por causa da biblioteca utilizada ser capaz de interpretar os nomes de cada campo de uma string JSON e deserializer a mesma numa estrutura, o outro fator que possibilita esta função é o operador «’a>"no titulo da estrutura e do operador "Option"nos campos das estruturas. Abaixo está um breve exemplo de uma estrutura gerada. Note-se que como usamos a biblioteca serde-json para serializar e deserializar mencionamos antes de cada estrutura que se deve usar a mesma para serializar e deserializar.

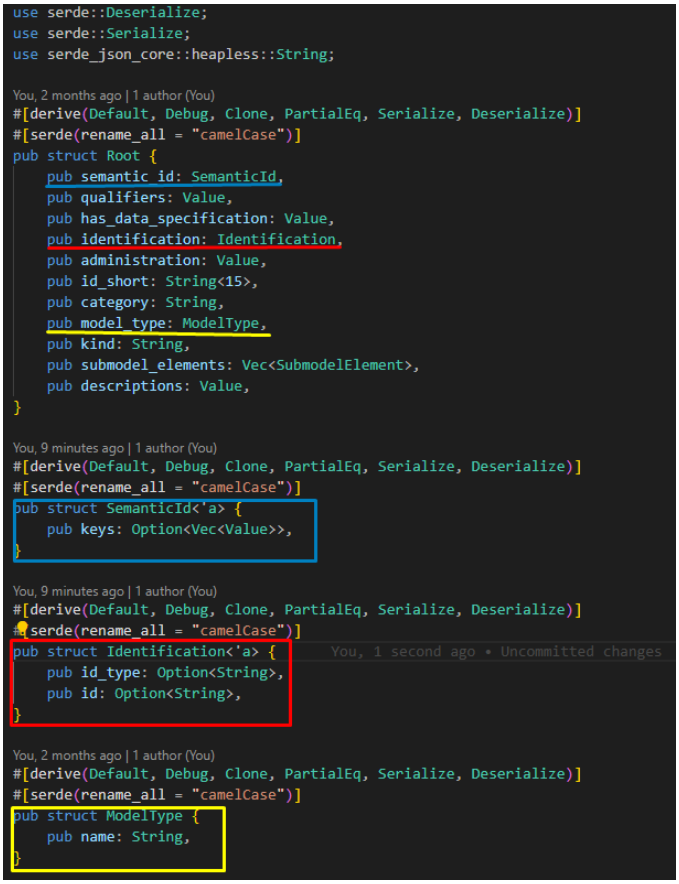

Figura 3.6: Exemplo de uma estrutura RUST gerada.

Analisando a figura 3.6 temos que para cada linha sublinhada a cores, temos de seguida delimitado por um retangulo da mesma cor a estrutura correspondente. Neste caso, a estrutura inicial é a primeira a ser declarada sendo que as estruturas quer pertencem à estrutura inicial são declarados apenas depois de declarar a estrutura inicial, esta sintaxe faz com que o comportamento do gerador seja mais simples e assim mais fácil de programar.

A simplicidade do Rust e os programas eficaz e leves que se podem criar em rust foram as razões pelas quais se começou por adotar uma implemenação do gerador de código que gerasse estruturas em Rust.

No entanto, Rust para dispositivos embutidos (No-std Rust) tem também limitações nas bibliotecas disponíveis e nas funções disponíveis inicialmente. Este obstaculo foi ultrapassado quando encontramos uma biblioteca no-std capaz de deserializar e serializar objetos e strings, a serde-json. No entanto à medida que se tentaria implementar uma API personalizada às estruturas foi cada vez mais dificil conseguir arranjar uma forma de respeitar os limites do no-std Rust devido ao suporte limitado dado ao ambiente no-std. Por fim, o que fez esta implementação inevitavelmente cair por terra foi impossibilidade de

conseguir ter um web interface para testar uma simples API gerada (Esta API apenas tinha o método INSERT e GET para a estrutura inicial, sendo que ignorava as estruturas das coleções de subelementos devido a dificuldades no controlo da memória usada durante a execução do programa).

Com estas dificuldade mudamos o foco do desenvolvimento do gerador para utilizar uma linguagem com um melhor suporte e maior desenvolvimento para dispositivos embutidos, resultando na implementação definitiva que gera estruturas em C++.

3.2.3.2 Gerador de código C++

Depois da implementação inicial do gerador de estruturas Rust, o periodo inicial para o desenvolvimento de um gerador de código C++ foi muito menor porque foi possivel reaproveitar o gerador inicial para construir o novo. Este gerador tal como o gerador inicial é um programa python que analisa o submodelo em formato JSON e gera estruturas de acordo.

Tal como no gerador anterior tinhamos encontrado uma biblioteca para serializar e deserializar, também aqui para C++ foi encontrada uma biblioteca capaz de fazer o mesmo, a biblioteca cJSON[8]. Esta biblioteca é utilizada nos métodos INSERT de cada estrutura, porque a atribuição automática de dados nas estruturas é feito através de strings Json interpretadas por esta biblioteca.

Este gerador mantém a capacidade de interpretar mensagens JSON parcias e ainda caso seja decidido um utilizador pode nem utilizar os métodos gerados e apenas inserir os dados manualmente. Este gerador tem uma particularidade quando gera estruturas e métodos devido á forma como o código C++ é compilado. Em C++, devemos sempre começar a declarar as estruturas que não dependem de nenhumas outras estruturas avançado apenas para a estrutura mãe apenas depois de declarar todas as estruturas filhas e o mesmo processo deve ser feito para métodos que dependam de outros métodos. É verdade que podemos evitar ter o cuidado de gerar as estruturas se antes de gerar a implementação das mesma colocassemos os protótipos de cada uma, no entanto, isto iria aumentar o tamanho do ficheiro que contém as estruturas e iriamos ter que ter os mesmos cuidados com os prototipos o que so torna a existencias dos mesmos redundantes de um ponto de vista da simplicidade do gerador e um aumento desnecessário ao tamanho do ficheiro.

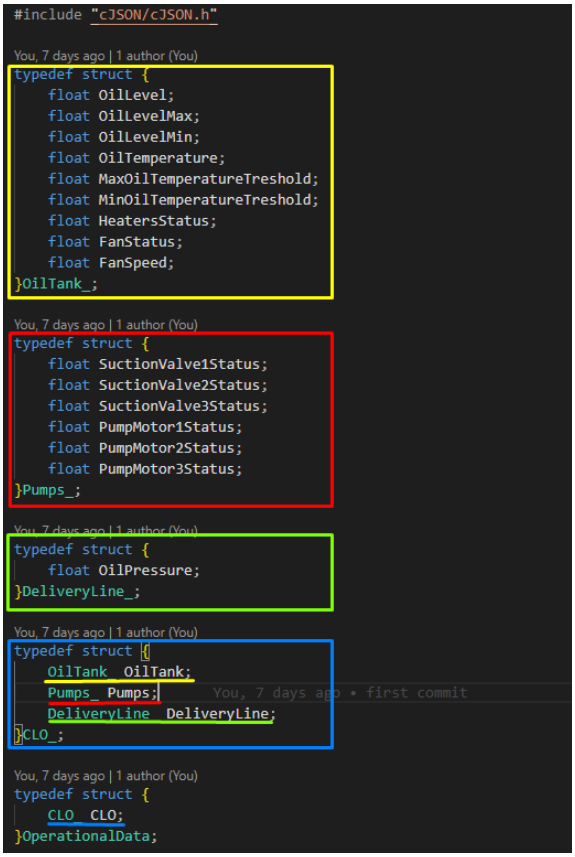

Figura 3.7: Exemplo de uma estrutura C++ gerada.

A figura 3.7 demonstra como as primeiras estruturas a serem declaradas são as estruturas que não dependem de outras estruturas. Na imagem cada estrutura gerada tem um retângulo à sua volta e caso seja mencionada noutra estrutura temos uma linha a chamar essa estrutura sublinhada com a cor correspondente. Note-se que no topo da figura 3.7 está o comando que inclui a biblioteca cJSON.

Para os métodos INSERT, o gerador gera sempre um método inicial do tipo Nomedo-submodelo com o nome CreateSubmodel que recebe como parâmetro de entrada um char* que será string em formato JSON e o objeto onde vamos inserir os dados. Depois gera métodos do tipo Nomedacoleção_ com o nome CreateNomedacoleçãosubmodel e recebe como parametros de entrada o elemento do submodelo em questão (a varíavel submodelname) e um objeto do tipo cJSON*, este objeto é uma string que está a ser interpretada pela biblioteca cJSON. Assim, para obter os dados da string JSON utiliza-se o comando cJSON_Parse para traduzir uma string num objeto cJSON, este comando apenas é executado no método INSERT da estrutura principal, depois disto usa-se o comando cJSON_GetObjectItemCaseSensitive para retirar a informaçao necessária para o campo da estrutura a ser processado, caso este campo seja um apontador para outra estrutura

vamos chamar o método INSERT referente a essa estrutura, caso contrário inserimos a informação extraída no campo respetivo.No fim do método INSERT do submodelo o último comando antes de retornar o objeto apagamos o objeto cJSON utilizado porque já não tem utilidade. Na imagem abaixo verificamos como os diferente métodos INSERT funcionam e interagem uns com outros. Cada método tem um retângulo colorido á sua volta e cada vez que é chamado noutro método sublinha-se a linha com a cor correspondente.

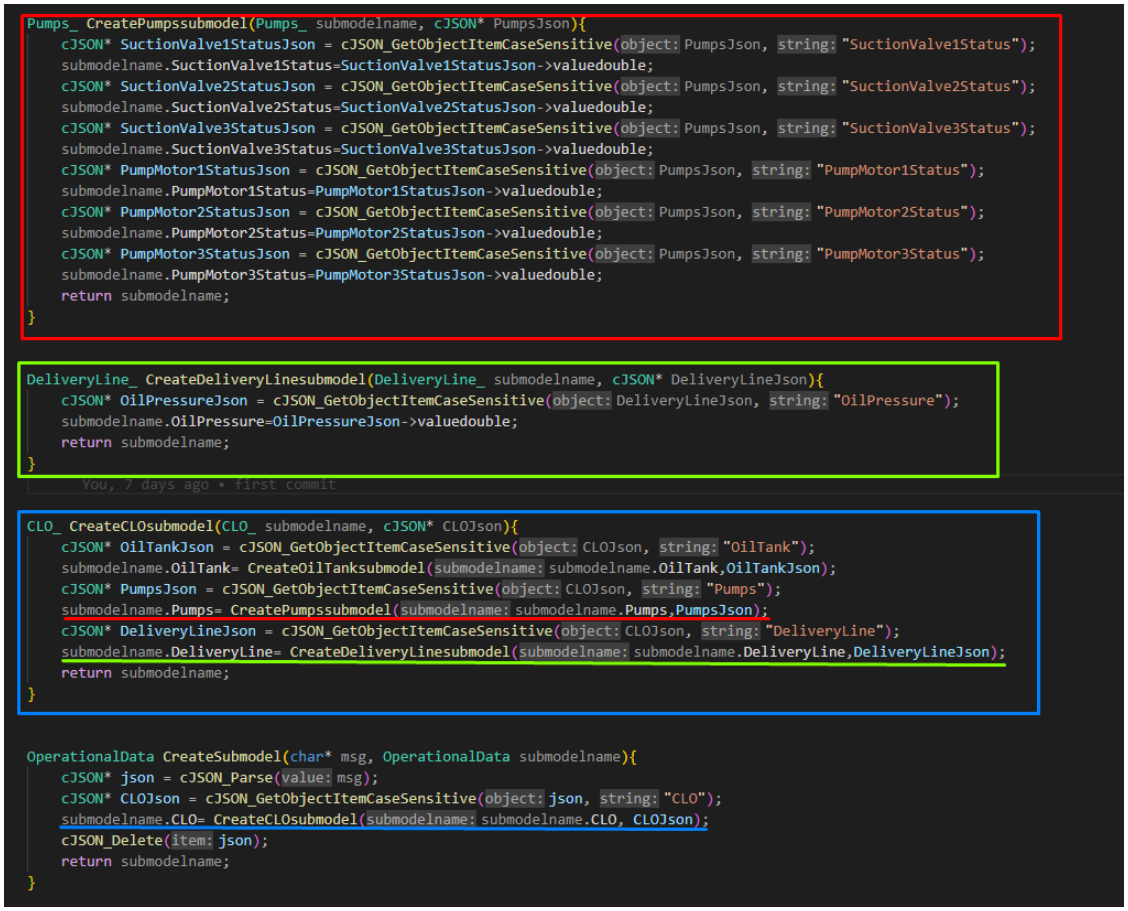

Figura 3.8: Exemplo de métodos INSERT C++ gerada.

Para os métodos GET, o gerador gera sempre métodos do tipo char* com o nome GetNomedacoleçãosubmodel e recebe apenas como parametro de entrada o submodelo que estamos a querer extrair informação. O produto final destes métodos é uma string em formato JSON de submodelo que serviu de parâmetro de entrada. Para formar a string inicialmente usa-se o comando JSON_CreateObject para criar um objeto cJSON, á medida que vai percorrendo os diferentes campos da estrutura usa-se o comando cJSON_AddNumberToObject para adicionar o campo e a sua informação ao objeto, caso o objeto seja uma coleção de subelementos não se usa o comando anterior, chama-se o método GET associado á coleção de subelementos encontrada. Depois de se percorrer todos os campos da estrutura converte-se o objeto cJSON numa string com o comando

cJSON_Print e o método GET retorna esta string. Abaixo está uma imagem onde tal como na imagem anterior utilizamos um código de cores para identificar as diferentes dependências entre os métodos.

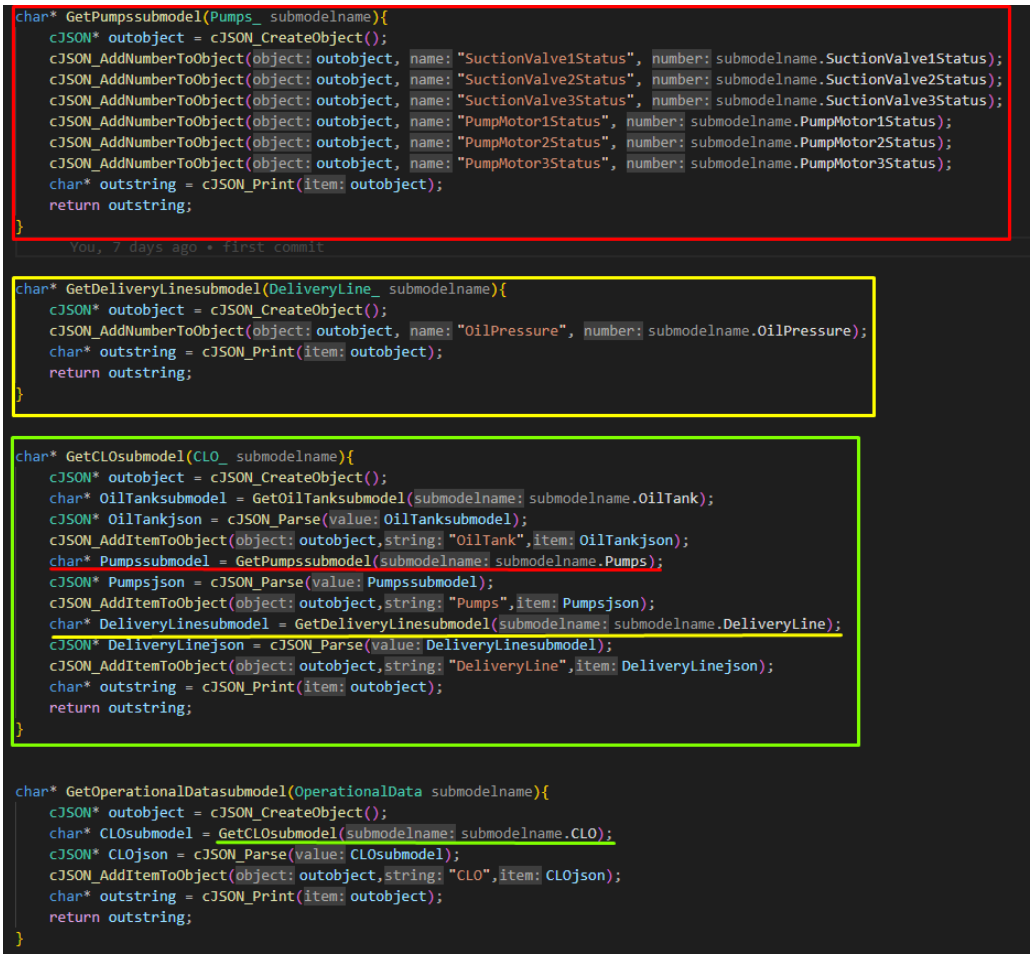

Figura 3.9: Exemplo de métodos GET C++ gerada.

3.2.4 API

A API do e-NOVAAS já foi explicada na secção anterior, trata-se dos métodos INSERT e GET das estruturas geradas, como estes métodos também são gerados pelo gerador de código eles são customizados para cada estrutura gerada sendo que para os métodos INSERT temos sempre uma função para a estrutura principal do submodel com o cabeçalho CreateSubmodel, caso existam coleções de subelementos dentro do submodelo gera-se métodos INSERT específicos para esta coleção, cada um destes métodos chama-se sempre CreateNomedacoleçãosubmodel, para uma coleção chamada SubelementCollection o método para inserir informação nesta coleção chama-se CreateSubelementCollectionsubmodel. Todos estes métodos retornam sempre o objeto do tipo de coleção ou submodelo onde se está a inserir informação.

Os métodos GET de cada estrutura também são gerados de acordo com a quantidade de submodelos e coleções de subelementos que o gerador identifica, sendo que para estes métodos temos sempre uma função que retorna uma string sob o formato JSON com o nome GetNomedacoleçãosubmodel, este cabeçalho serve tanto para submodelos ou coleções de subelementos.

3.2.5 Northbound Interface

Este componente não é um componente fixo do e-NOVAAS no sentido em que pode ser implementado um northbound interface diferente consoante a escolha do utilizador. Para efeitos de demonstração, durante a testagem do código gerado utilizou-se um template em simulador Wokwi de um servidor Web em localhost que interage com a API para obter ou inserir informação ou objetos de acordo com as estruturas geradas. Este servidor tem uma interface gráfica que consoante o input do utilizador pode chamar os métodos INSERT ou GET do submodelo ou então os métodos GET das coleções de subelementos.

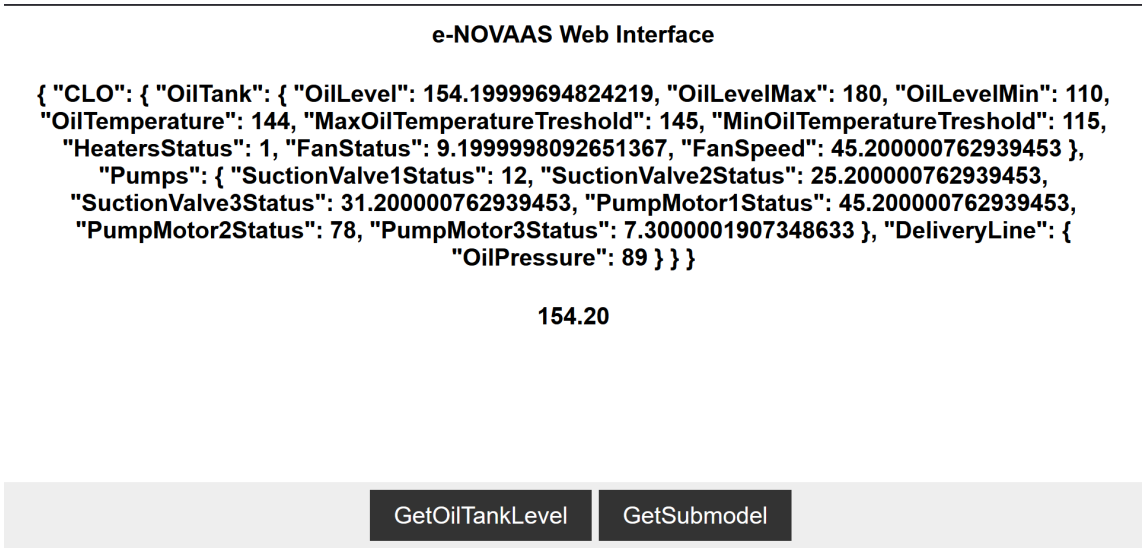

Figura 3.10: API Web Interfacce.

Obeservando a figura 3.10 verificamos que temos botões que chamam método GETSubmodel gerado e também acedem diretamtente ao elemento OilLevel da coleção OilTank que por sua vez pertence á coleção CLO que pertence á estrutura principal, quando pressionamos estes botões o servidor respondem com uma mensagem em formato JSON do submodelo ou coleção de subelementos conforme o que for chamado pelo utilizador. Por agora está ser utilizada uma interface gráfica fixa e não customizada para as estruturas e métodos gerados. talvez numa implementação futura podemos usar o gerador de código

para também gerar uma página HTML com uma interface gráfica customizada para o código gerado, mas por agora e para efeitos de testagem temos que escrever manualmente a página HTML. Outra implementação futura pode ser uma interface capaz de chamar os métodos INSERT para além dos métodos GET, modificando a interação do Northbound Interface para uma interação bidirecional, diferente do que se verifica na figura 3.2.

4

Testes

Neste capitulo vai ser testado com três casos distintos como se comporta o e-NOVAAS e todos os componentes gerados. Estes três casos vão ter cada um níveis de complexidade distintos.

No capitulo anterior, definimos que uma coleção de subelementos equivale a uma nested structure em C, uma estrutura dentro de outra, este conceito acrescenta complexidade aos submodelos utilizados no e-NOVAAS. Esta complexidade pode ser avaliada como graus de profundidade, por exemplo uma estrutura sem nested structures vai ter um grau de profundidade, uma estrutura com uma nested strutcture tem dois graus de profundidade, uma estrutura com uma nested structure que contenha outra nested structure vai ter três graus de profundidade e assim em diante.

Desta forma, o grau de profundidade de uma estrutura é uma forma de medir a sua complexidade e pode servir para avaliar o código gerado. Outro aspeto a considerar para testar o código gerado é a variedade de elementos no submodelo, uma grande diversidade nos tipos de elementos utilizados significa que uma estrutura é composta por diferentes tipos de dados, inteiros, caracteres, double, floats, strings, entre outros.

Para testar as estruturas e métodos gerados, vamos mostrar uma Web Interface para cada caso, e iremos chamar o método GetSubmodel da estrutura gerada e um método GET para uma coleção de Subelementos caso a estrutura tenha uma, iremos ainda chamar um elemento individual da estrutura sem utilizar métodos GET. Mais uma vez deve ser sublinhado que todos os métodos e estruturas foram gerados pelo gerador de código e que apenas a ligação dos mesmo á Web Interface (Northbound Interface) foi programado manualmente.

4.1 Primeiro Caso

Inicialmente vamos testar com uma estrutura simples com apenas um grau de profundidade, composto por quatro elementos. Esta estrutura foi gerada tendo como base o submodelo de um sensor de Temperatura e Humidade DHT22. É composta por um campo correspondente ao nome do sensor, um campo que serve como número de identificação do sensor, uma campo que armazena os valores da Temperatura medidos e um último campo armazenar os valores de Humidade medidos. Na figura 4.1 abaixo está a estrutura gerada.

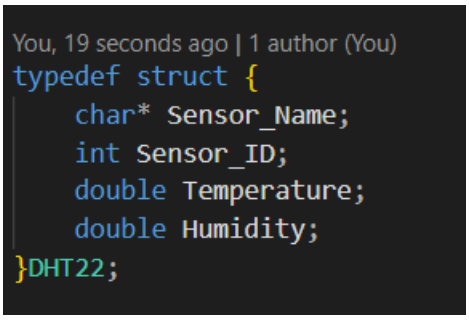

Figura 4.1: Estrutura Utilizada para o primeiro caso

Visto que estamos a simular um Esp32 em Wokwi podemos adicionar alguns componentes ao ambiente tal como se estivessesmos a adicionar componentes a uma placa real e já que se vai usar o submodelo de um DHT22 podemos adicionar este sensor à simulação do Wokwi, o que nos permite testar como o e-NOVAAS se poderá comportar num ambiente real.

Assim, foi necessário usar a biblioteca do DHT22 para conseguir comunicar com o sensor corretamente, depois disto basta apenas afetar o objeto da estrutura com os valores de Temperatura e Humidade que o sensor vai registando nos campos adequados como se verifica na imagem 4.2 abaixo:

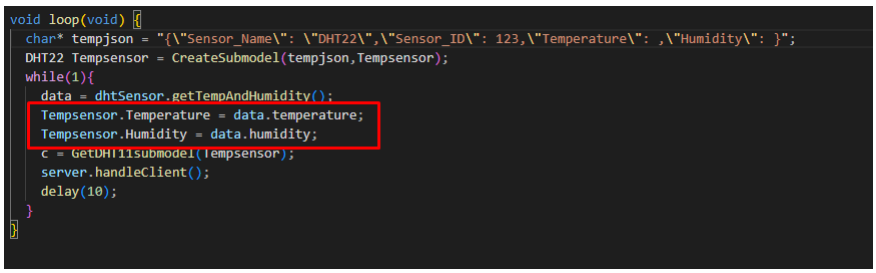

Figura 4.2: Codigo usado para associar o sensor com a estrutura gerada

Tendo o escrito o código basta apenas testar a resposta da api com os dados de temperatura e humdidade simulados pelo Wokwi. Assim, o esquema do Wokwi vai ser o que está demonstrado na imagem 4.3 abaixo.

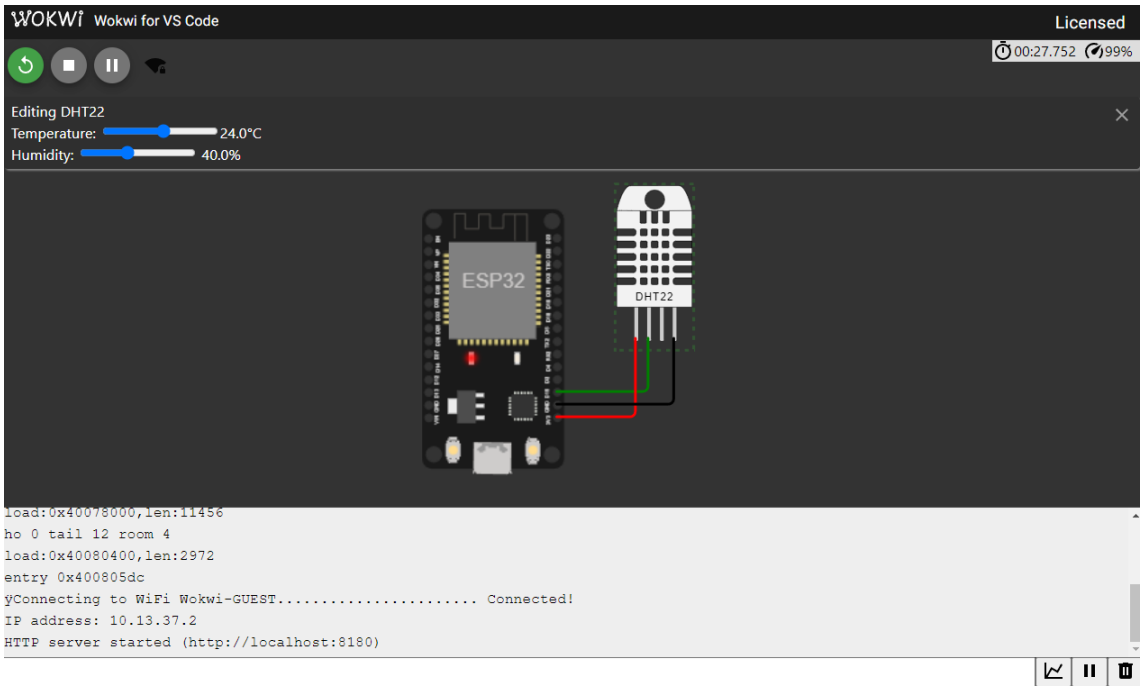

Figura 4.3: Esquema de Wokwi utilizado para simular o caso com uma DHT22

Note-se que o simulador foi configurado para registar nos sensores 24ºC de temperatura e 40% de humidade. O teste será bem sucedido se a API mostrar mensagens JSON onde a temperatura a e humidade tenham estes valores também. E como se pode verificar na imagem 4.4 abaixo a recolha da informaçao de temperatura e humidade e consequente envio como mensagem JSON funciona.

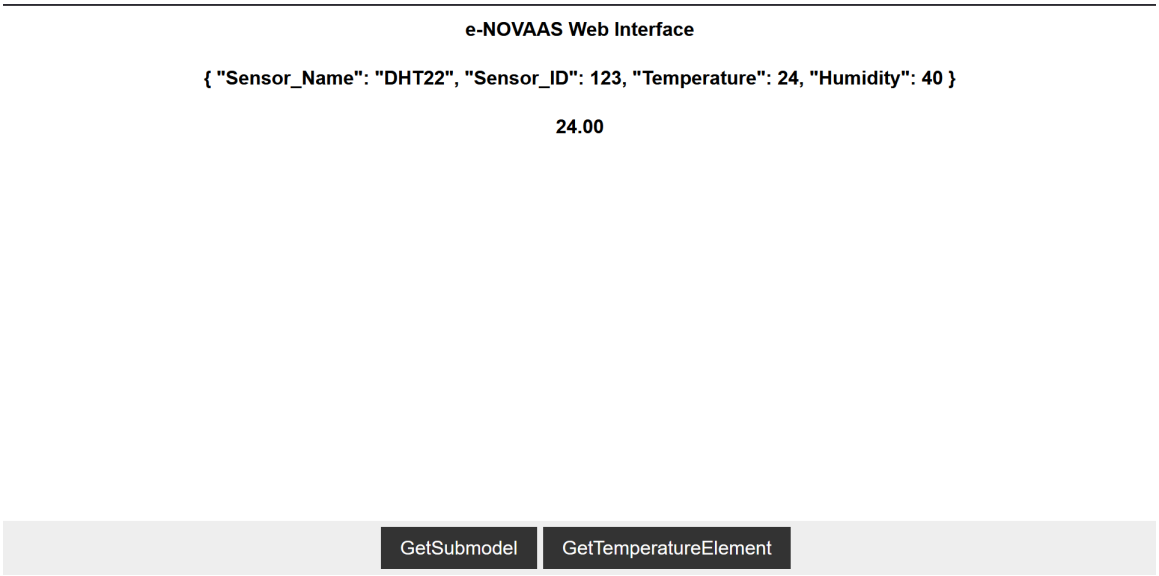

Figura 4.4: Resposta da Interface Web para o caso com o sensor DHT22

Em termos de memória, o programa compilado ocupará 38,7 Kb de memória RAM e 765,7 Kb de memória Flash. Apenas comparando com os próximos casos será possível concluir se a complexidade das estruturas geradas irá afetar bastante a quantidade de memória utilizada. Algo que podemos concluir no entanto, é que o esp32 ou placas semelhante a um esp32 será capaz de executar este programa, desde que se tenha cuidado com apontadores e alocação de memoria durante a execução, visto que quando compilado ocupamos cerca de 12% da RAM e cerca de 58% da memoria Flash do esp32.

```
RAM:   [=         ]  11.8% (used 38728 bytes from 327680 bytes)
Flash: [======    ]  58.4% (used 765730 bytes from 1310720 bytes)
```

Figura 4.5: Memória Utilizada quando o programa é compilado

4.2 Segundo Caso

Para este segundo caso de testes vamos testar a recolha e envio de informação outra vez, mas neste caso vamos gerar uma estrutura segundo um submodelo criado para a demo do NOVAAS. Este submodelo representa vários campos e estados de bombas e valvulas.

O submodelo vai ser mais complexo que o submodelo do caso anterior, sendo composto por várias estruturas dentro de estruturas, assim o gerador irá criar as seguintes estruturas dependentes umas das outras.

```
You, last month | 1 author (You)
typedef struct {
    float OilLevel;
    float OilLevelMax;
    float OilLevelMin;
    float OilTemperature;
    float MaxOilTemperatureTreshold;
    float MinOilTemperatureTreshold;
    float HeatersStatus;
    float FanStatus;
    float FanSpeed;
}OilTank_;

You, last month | 1 author (You)
typedef struct {
    float SuctionValve1Status;
    float SuctionValve2Status;
    float SuctionValve3Status;
    float PumpMotor1Status;
    float PumpMotor2Status;
    float PumpMotor3Status;
}Pumps_;

You, last month | 1 author (You)
typedef struct {      You, last month
    float OilPressure;
}DeliveryLine_;

You, last month | 1 author (You)
typedef struct {
    OilTank_ OilTank;
    Pumps_ Pumps;
    DeliveryLine_ DeliveryLine;
}CLO_;

You, last month | 1 author (You)
typedef struct {
    CLO_ CLO;
}OperationalData;
```

Figura 4.6: Estrutura gerada do submodelo do segundo caso

Neste caso, não temos um sensor que possamos simular no Wokwi, como tal, vamos inicializar a estrutura do submodelo com uma mensagem JSON para preencher toda a estrutura, de seguida vamos atribuir a uma variavel A o retorno da função GET da estrutura inteira, outra (variável B) irá ter o retorno da função GET da sub-estrutura "Oil Tank", por último vamos atribuir a uma variável C o campo "Oil Level"da sub-estrutura "Oil Tank".

```
void loop(void) {
  char* json = "{\"CLO\":{\"OilTank\": {\"OilLevel\":154.2,\"OilLevelMax\": 180,\"OilLevelMin\": 110,\"OilTemperature\": 144,\"MaxOilTemper
  OperationalData OpData = CreateSubmodel(json, OpData);
  while(1){
    a = GetOperationalDatasubmodel(OpData);
    b = GetOilTanksubmodel(OpData.CLO.OilTank);
    c = OpData.CLO.OilTank.OilLevel;
    server.handleClient();
    free(a);
    free(b);
    delay(10);
  }
}
```

Figura 4.7: Void Loop onde os valores de temperatura e humidade são colocado em tempo real na estrutura

Quando o código for compilado e executado vai criar uma interface Web com três botões onde iremos chamar cada uma das variaveis A,B,C referidas anteriormente. Na imagem abaixo vemos como a Web Interface responde quando pressionamos todos os botões, daí existir uma mensagem JSON do submodelo todo o que corresponde á informação presente na varíavel a, uma outra mensagem JSON da sub-estrutura "Oil Tank"o que corresponde á varíavel b e por fim uma campo que irá corresponder á informação do "Oil Level"que está presente na varíavel c.

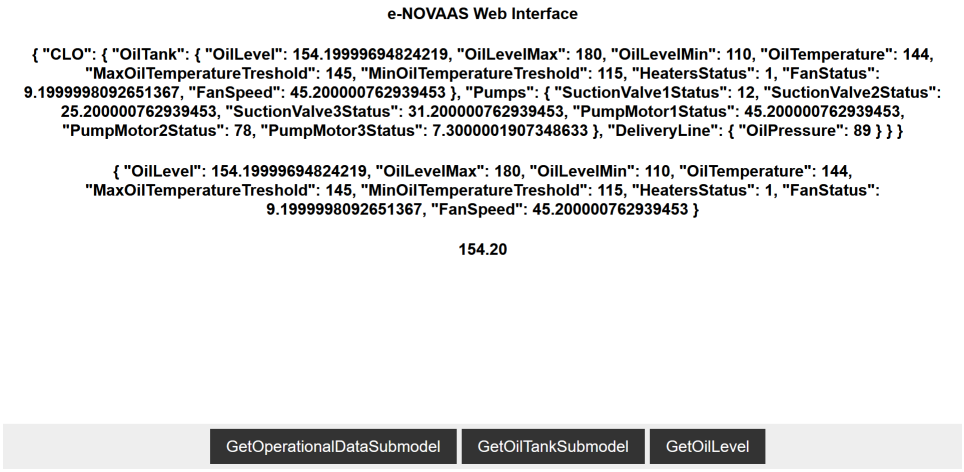

Figura 4.8: Resposta da Interface Web para o caso com o submodelo retirado dos exemplos do NOVAAS

Algo que é importante notar é que este programa quando compilado vai ocupar 767Kb de memória Flash . Em termos de memória Flash nota-se que este caso ocupa mais 2Kb do que o caso anterior, provavelmente devido á maior complexidade deste caso, este aumento também pode estar relacionado com o tamanho das estruturas e métodos porque esse ficheiro ocupa agora 8Kb de espaço. Em termos de memória RAM este caso ocupa 38,7kb de memória, um valor semelhante ao caso anterior, talvez porque estamos nos dois casos

apenas a lidar com um objeto de cada vez e a diferença nos tamanhos da estrutura em RAM não se torna tão evidente como na memória Flash.

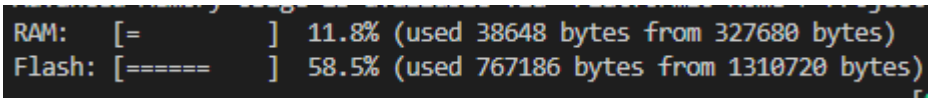

Figura 4.9: Impacto do programa compilado na memória RAM e Flash do Esp32

4.3 Terceiro Caso

Para o terceiro e último caso dos Testes vai ser utilizado o submodelo mais complexo até agora, este submodelo foi retirado diretamente dos exemplos disponíveis no site da IDTA [21]. Este submodelo vai representar uma nameplate que contém várias informação como por exemplo o nome do fabricante da mesma (Manufacturer Name), o numero de serie (Serial Number), o ano de Fabríco (Year of Construction) e a morada (Physical Address), esta última é uma coleção de subelementos e como tal quando for interpretada vai ser uma nested strucuture na estrutura principal. Abaixo está representado o submodelo utilizado da nameplate no aasx-package-explorer.

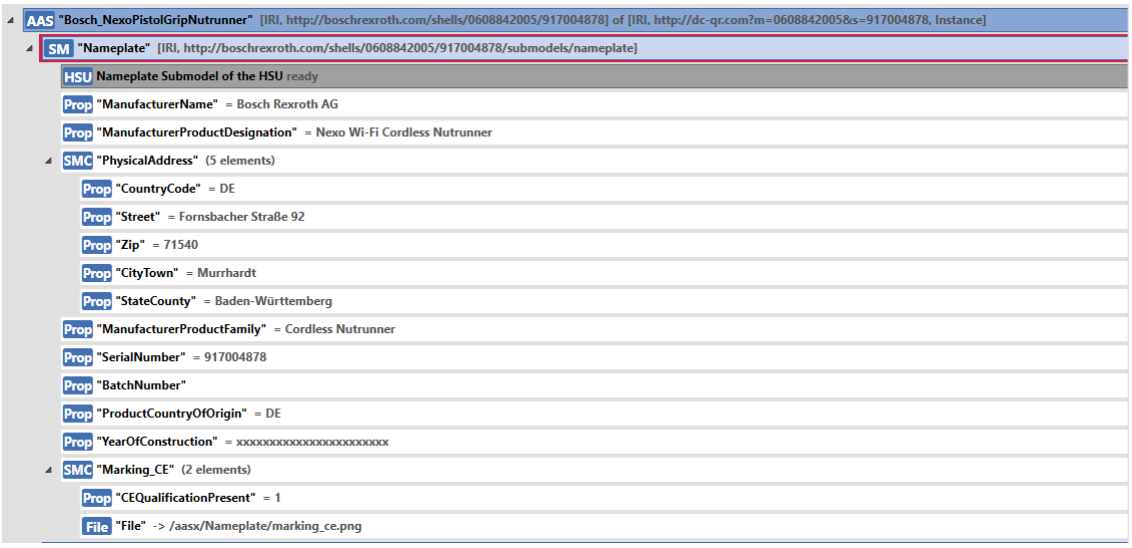

Figura 4.10: Estrutura do submodelo Nameplate no aasx-package-explorer

Este submodelo contém alguns elementos que reprensentam mensagens e strings de informação, estas varíaveis vão acrescentar á complexidade da estrutura gerada porque para os métodos GET e INSERT gerados devem ter este pormenor em conta.

Figura 4.11: Estruturas geradas através do submodelo Nameplate

Como existem strings nas estruturas, os métodos GET e INSERT têm alguns cuidados a manipular strings, cuidados como a utilização de strdup para a duplicação de strings em vez de uma simples cópia de varíaveis, as funções geradas também irao ter o cuidado de apagar quaisquer apontadores depois de esgotada a sua utilidade, na figura abaixo está o exemplo da função gerada CreateSubmodel.

```
Nameplate CreateSubmodel(char* msg, Nameplate submodelname){
    cJSON* json = cJSON_Parse(value: msg);
    submodelname.ManufacturerName=strdup(String: cJSON_GetObjectItemCaseSensitive(object: json, string: "ManufacturerName")->valuestring);
    submodelname.ManufacturerProductDesignation=strdup(String: cJSON_GetObjectItemCaseSensitive(object: json, string: "ManufacturerProductDesignation")->valuestring);
    cJSON* PhysicalAddressJson = cJSON_GetObjectItemCaseSensitive(object: json, string: "PhysicalAddress");
    submodelname.PhysicalAddress= CreatePhysicalAddresssubmodel(submodelname: submodelname.PhysicalAddress, PhysicalAddressJson);
    submodelname.ManufacturerProductFamily=strdup(String: cJSON_GetObjectItemCaseSensitive(object: json, string: "ManufacturerProductFamily")->valuestring);
    submodelname.SerialNumber=strdup(String: cJSON_GetObjectItemCaseSensitive(object: json, string: "SerialNumber")->valuestring);
    cJSON* BatchNumberJson = cJSON_GetObjectItemCaseSensitive(object: json, string: "BatchNumber");
    submodelname.BatchNumber=BatchNumberJson->valuedouble;
    submodelname.ProductCountryOfOrigin=strdup(String: cJSON_GetObjectItemCaseSensitive(object: json, string: "ProductCountryOfOrigin")->valuestring);
    submodelname.YearOfConstruction=strdup(String: cJSON_GetObjectItemCaseSensitive(object: json, string: "YearOfConstruction")->valuestring);
    cJSON* Marking_CEJson = cJSON_GetObjectItemCaseSensitive(object: json, string: "Marking_CE");
    submodelname.Marking_CE= CreateMarking_CEsubmodel(submodelname: submodelname.Marking_CE, Marking_CEJson);
    cJSON_Delete(item: json);
    return submodelname;
}
```

Figura 4.12: Função CreateSubmodel gerada do submodelo Nameplate

Tal como no caso anterior iremos utilizar uma mensagem JSON declarada no inicio do programa para preencher a estrutura e depois iremos pegar em três variaveis para enviar

para a API uma mensagem JSON do submodelo, através da função GetNameplatesubmodel, outra mensagem JSON para enviar uma coleção de subelementos da Physical Address da nameplate, com a função GetPhysicalAddresssubmodel, por fim iremos aceder diretamente ao campo Street da coleção de subelementos da Physical Address da Nameplate. Abaixo está o void loop utilzado para realizar o teste.

```
void loop(void) {
  char* json_message = "{\"ManufacturerName\":\"Example Manufacturer\",\"ManufacturerProductDesignation\":\"Product XYZ\",
  Nameplate Bosch = CreateSubmodel(json_message,Bosch);
  while(1){
    a = GetNameplatesubmodel(Bosch);
    b = GetPhysicalAddresssubmodel(Bosch.PhysicalAddress);
    c = strdup(Bosch.PhysicalAddress.Street);
    server.handleClient();
    free(a);
    free(b);
    free(c);
  delay(10);
  }
}
```

Figura 4.13: VoidLoop onde o submodelo é atualizado em tempo real

Depois compilado e executado o programa, a API surge com três botões que irão chamar as três variaveis mencionadas anteriormente, o resultado é bem-sucedido e acaba por ser duas mensagens JSON corretamente transmitidas e que caso seja preferível é sempre possível aceder ao campo da um elemento da estrutura diretamente sem utilizar um método específico

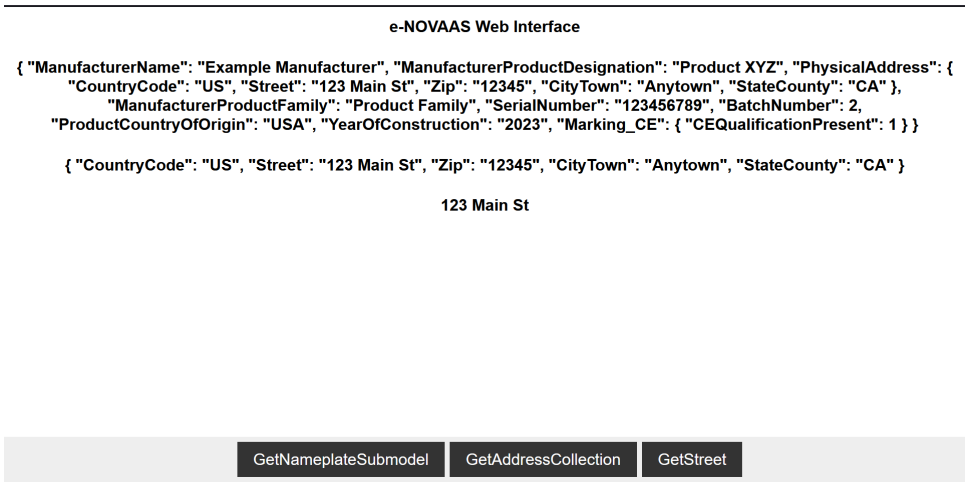

Figura 4.14: Resposta do Interface Web para o tercerio caso

Pro fim, se analisarmos o espaço que este caso ocupa em memória Flash e memória RAM verifica-se que ocupa respetivamente, 765Kb e 38,6Kb. O ficheiro structs.c onde está a implementação de todos os métodos e estruturas geradas ocupa neste caso 6Kb. Quando comparado com o caso anterior repara-se que o caso atual ocupa ligeiramente menos espaço que o caso anterior, visto que a maior diferença entre os dois caso é o submodelo utilizado e as estruturas geradas consequentes. Assim, uma conclusão que possa justificar esta diminuição do espaço utilizado pode ser porque como este caso usa um submodelo publicado com a aprovação da IDTA este submodelo pode estar melhor otimizado do que o submodelo desenhado para o NOVAAS.

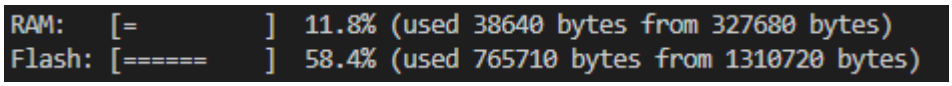

Figura 4.15: Impacto que o programa compilado para este caso tem na memória Flash e RAM do Esp

5

Aplicação no Mundo Real: Monitorização de um campo agrícola

5.1 Introdução/Contextualização

Num contexto agrícola, a integração de uma implementação de um AAS pode representar uma evolução significativa na gestão das operações agrícolas, especialmente se o AAS for fácil e simples de instalar na operação como é o caso para uma implmentação embutida onde se pode utilizar microcontroladores. No coração desta evolução encontra-se a capacidade de monitorizar ativos físicos, como sensores de temperatura e humidade. Estes sensores são estrategicamente instalados nas zonas do campo para recolher dados críticos sobre as condições ambientais, e os AAS desempenham o papel de elo de ligação entre as condições do campo e os sistemas de gestão agrícola.

Sendo assim, vamos então introduzir o e-NOVAAS num ambiente real. Considera-se um pequeno campo agrícola, o dono do campo pretende saber em tempo real os níveis de Temperatura e Humidade do campo ao longo do dia.

Figura 5.1: Fotografia retirada do local do campo onde o AAS vai recolher os dados de Temperatura e Humidade

5.2 Setup

Como foi dito anteriormente o AAS vai servir como o elo de ligação entre as condições do campo e o sistema de gestão agrícola. Para estabelecer esta ligação vamos necessitar de construir um submodelo. Vai ser utilizado um submodelo de um digital twin de um arduino ligado a um sensor DHT22, este submodelo é idêntico ao submodelo estudado no subcapitulo 4.1. O submodelo vai ser constituido por 4 elementos, um referente ao nome do sensor, um referente ao ID do sensor e os restantes dois serão referentes á Temperatura e Humidade.

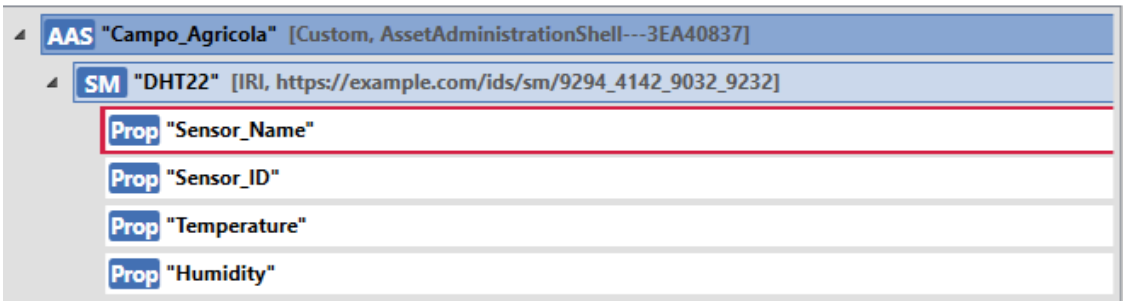

Figura 5.2: Submodelo Utilizado no campo agrícola

Depois de formulado o submodelo basta exportar o submodelo em formato JSON e utilizar o gerador de código para gerar a estruturas e os métodos para serem utilizados no microcontrolador que irá fazer a monitorização da temperatura. O microcontrolador que vai utilizado vai ser um Arduino Uno genérico, como esta placa não tem uma placa WIFI, não é possível criar uma interface Web, a alternativa encontrada foi então transmitir a mensagem com a temperatura e Humidade por porta série. Na imagem abaixo, está a estrutura gerada com o submodelo da figura 5.2.

```
You, 19 seconds ago | 1 author (You)
typedef struct {
    char* Sensor_Name;
    int Sensor_ID;
    double Temperature;
    double Humidity;
}DHT22;
```

Figura 5.3: Estrutura gerada

Depois de utilizar o gerador de código basta criar um ficheiro .uno para que possamos carregar a estrutura para o arduino. Utilizando a biblioteca do sensor DHT disponível para o arduino conseguimos extrair o valor de Humidade e TemperaturaPara inserir na estrutura até podemos apenas afetar os campos da estrutura referentes à Temperatura e Humidade depois de gerarmos o objeto da estrutura. Como queremos transmitir a estrutura via porta série, basta utilizar o comando Serial.print para enviar a mensagem. Desta forma, o código que vai ser carregado para a placa está na imagem abaixo.

```
#include "customstruct/cJSON/cJSON.h"
#include <string.h>
#include "customstruct/struts.c"
#include "DHT.h"

#define DHTPIN 2
#define DHTTYPE DHT11   // DHT 11

DHT dht(DHTPIN, DHTTYPE);
void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  Serial.println(F("DHTxx test!"));
  dht.begin();

}

void loop() {
  char* tempjson = "{\"Sensor_Name\": \"DHT22\",\"Sensor_ID\": 123,\"Temperature\": ,\"Humidity\": }";
  dHT22 Tempsensor = CreateSubmodel(tempjson, Tempsensor);
  while (1) {
    // put your main code here, to run repeatedly:
    float h = dht.readHumidity();
    // Read temperature as Celsius (the default)
    float t = dht.readTemperature();
    Tempsensor.Temperature = t;
    Tempsensor.Humidity = h;
    Serial.print("This is temp:");
    Serial.println(t);
    char* c = GetDHT22submodel(Tempsensor);
    Serial.println(c);
    delay(10);
  }
}
```

Figura 5.4: Código que vai ser carregado

Depois de fazer isto, basta apenas montar o sensor DHT no Arduino UNO, como está na figura abaixo.

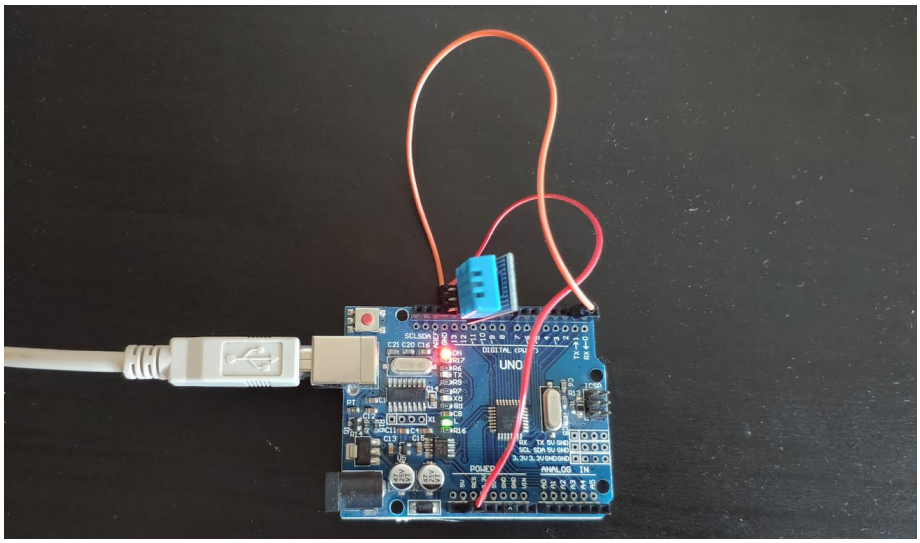

Figura 5.5: Placa Arduino que foi utilizada já com o sensor DHT montado

Para certificar que tudo funcionava antes ir para o campo, testou-se o arduino e verificou-se que tudo estava funcional. A imagem abaixo é a mensagem que o arduino irá enviar para a porta série.


```
{
        "Sensor_Name":  "\u0015\u001a\u0016\u0016",
        "Sensor_ID":    773,
        "Temperature":  29,
        "Humidity":     38
}
```


Figura 5.6: Mensagem recebida na porta série do computador. Nota: este valor não foi medido no campo agricola.

5.3 Instalação da placa e recolha de dados

Agora que todos os passos de preparação foram feitos, vai se colocar a placa a funcionar no campo. A placa foi posta na erva do campo a registar a Temperatura e Humidade durante as 13h e as 16h num dia de céu limpo e o campo foi regado antes de colocar a placa. A placa irá enviar informação de temperatur e humidade a cada segundo. Como

estamos a transmitir para a porta série, o arduino está ligado a um computador que vai recolher a informação que o arduino enviar.

Depois de passadas as três horas no campo agrícola, o Arduino registou algumas variações da temperatura e Humidade, como se verifica na imagem do gráfico abaixo.

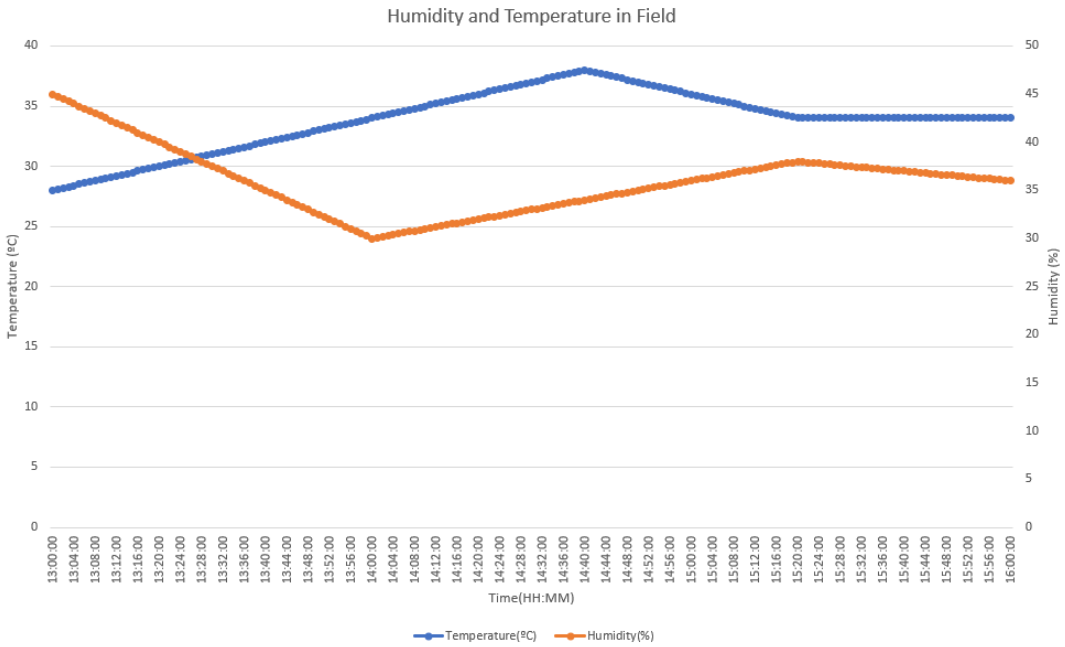

Figura 5.7: Gráfico da Temperatura e Humidade registados ao longo da temperatura.

Analisando os dados do gráfico da imagem 5.7, verificamos que a Temperatura variou entre os 28 e os 38 graus celsius, e a humidade variou entre os 44% e os 35%. Inicialmente estes valores podem parecer estranho mas podemos explicar a amplitude de temperatura medida porque o sensor teve momentos em que teve exposto diretamente ao sol o que pode causar um aquecimento exagerado. De qualquer forma, o que era importante era registar e enviar os dados e esse objetivo foi claramente atingido.

5.4 Analise de Resultados

Nesta secção final vamos fazer um balanço desta implementação no mundo real do e-NOVAAS. Obviamente esta implementação foi bem-sucedida, o objetivo de monitorizar e registar informação foi cumprido e com estes dados vai ser depois possível alterar os padrões da rega de forma a ter uma gestão mais eficiente e sustentável.

Como foi dito antes, utilizar um AAS para monitorizar este campo torna possível melhorar os padrões de rega. No entanto, também pode servir para prevenir ou antecipar o aparecimento de pragas como o mildio que afeta colheitas como o tomate e as batatas. Esta praga cresce especialmente em regiões com húmido ou com períodos prolongados de humidade e pode causar a destruição total de plantas. Com o e-NOVAAS a monitorizar constantemente o campo conseguimos detetar os períodos de maior humidade onde o mildio cresce.

Mas para além do objetivo principal, é importante sublinhar alguns aspetos importante dos benefícios de utilizar o e-NOVAAS, o investimento de tempo e dinheiro necessário para chegar a uma solução pronta a colocar no terreno.

Todo o processo de montagem e escrita do código foi cronometrado, foram gastos 12 minutos para criar o submodelo do DHT22, depois disto o processo de criação das estruturas foi instantâneo e automático graças ao gerador de código, após termos as estruturas demorou-se 20 minutos para preparar a biblioteca necessária para interagir com o DHT e escrever o código que foi carregado na placa, depois deste processo a placa está pronta a ser colocada no terreno. O processo no total demorou cerca de 32 minutos, o que demonstra como o e-NOVAAS pode apressar o processo de implementação de um AAS, caso utilizasse outras implementações que não fossem embutidas iriamos certamente demorar muito mais tempo porque para além de precisar de escrever o código necessário para implementar a solução também seria preciso aprender a utilizar outra implementação de AAS. Como o e-NOVAAS é capaz de gerar código automaticamente é muito mais rápido implementar um AAS no terreno, o que permite muito mais rápidamente adaptar um ambiente industrial para os standards da indústria 4.0.

6

Conclusão

6.1 Trabalho Futuro

A presente tese demonstra uma prova de conceito valiosa no contexto da tradução de submodelos de Digital Twins para código funcional em dispositivos com recursos limitados. Esta prova de conceito demonstra como é possivel facilitar o processo de implementação de novos digital twins, e como se pode rapidamente e facilmente adaptar casos simples de implementações como foi o exemplo do primeiro caso testado, e também adaptar casos mais complexos e mais exigentes como foram os exemplos do segundo e terceiro casos testados.

Como foi referido, a presente implementação do e-NOVAAS serve como uma prova de conceito apenas, como tal existe ainda alguns pontos que podem ser melhorados, assim é importante ponderar algumas possiveis melhorias e funcionalidades que poderiam ser implementadas no futuro.

Uma possivel melhoria é aumentar a variedade dos tipos de dados que o gerador consegue identificar. Atualmente, o gerador de código consegue distinguir entre apenas alguns tipos de dados de indústria 4.0, por exemplo, existe no aasx-package-explorer um tipo de dados chamado "date"este tipo de dados é capaz de representar uma timestamp, na versão atual do gerador de código este tipo de dados não iria ser reconhecido pelo gerador quando se fosse ler a exportação do submodelo o que iria resultar neste elemento ser representado apenas por um float porque essa é a opção default do gerador. Numa implementação futura, o gerador podia identificar o "date"e atribuir um tipo de dados semelhante a um timestamp ou a uma estrutura que represente horas, minutos, segundos, etc.

À medida que novos tipos de dados são considerados, outra melhoria que deve ser considerada será um melhor controlo da quantidade de memória utilizada. Isto irá permitir que dispositivos ainda mais limitados em termos de espaço sejam compatíveis com o e-NOVAAS, e para dispositivos com mais espaço poderiam até mesmo correr

mais submodelos do e-NOVAAS simultaneamente. Independente do sistema, uma maior eficiência em termos de RAM e memória flash ocupada é sempre algo a considerar para implementações futuras.

Uma funcionalidade que pode no futuro ser muito útil seria por exemplo, a existência de uma ligação direta entre o aasx-package-explorer e o gerador de código do e-NOVAAS. Será interessante ter uma opção de exportar submodelos e num passo só gerar o ficheiro que contém as estruturas e métodos gerados a partir do interface do aasx-package-explorer, isto iria facilitar o processo de implementação.

Para implemetações futuras também pode ser considerado extender a geração de código para o northbound interface. Se o gerador for capaz de automaticamente criar um interface Web customizado para a estrutura gerada. Durante os testes, foi necessário alterar a interface manualmente para corresponder á estrutura a ser utilizada, esta alteração não só demora algum para alguem que não tenha muita experiência com HTML mas também é muito repetitivo, especialmente se a quantidade de estruturas diferentes for muito grande.

6.2 Notas Finais

Para concluir esta dissertação vamos fazer um breve balanço geral. No inicio da dissertação identificou-se que existe muito pouco suporte de AAS para dispositivos embutidos, Estudou-se o que constitui um AAS e as várias implementações que existem atualmente.

Acercas das implementações atuais, fez-se um estudo que revelou alguma propriedades interessantes que se devia considerar para a implementação, propriedades como por exemplo a compatibilidade com submodelos personalizados que existe no FAAAST ou a criação de uma API como encontramos no FAAAST e no Basyx. O design da estrutura também foi inspirada na estrutura do NOVAAS onde temos um southbound interface para interagir com o asset e um northbound interface a partir do qual vamos enviar a informação do submodelo.

Depois de estudadas as outras implementações, desenvolveu-se o e-NOVAAS, é composto por um gerador de código e os códigos e métodos que gera. Este gerador é capaz de interpretar submodelos desenhados pela ferramenta utilizada para definir o standard da indústria, o aasx-package-explorer. A utilização deste gerador de código serve para garantir que em sistemas embedded estamos a utilizar uma implementação de referência.

Depois de desenvolvido o e-NOVAAS, foi realizado três testes cada um mais complexo que o anterior. Estes testes servem para demonstrar como o e-NOVAAS funciona com diferentes níveis de complexidade, serviram também para estudar como é a gestão de espaço de memória Flash e RAM. Nos casos testados, onde se utilizava um esp32 e o

e-NOVAAS comunicava através de um interface Web, a memória Flash utilizada nunca superou 59 % da capacidade do dispositivo e memória RAM nunca superou os 12 %.

Depois de feitos estes testes, inseriu-se o e-NOVAAS num contexto real onde muito rapidamente se desenvolveu uma solução capaz de monitorizar um campo agrícola. Este contexto serviu para reforçar o vasto leque de cenários onde o e-NOVAAS, neste caso serviu para monitorizar a temperatura e humidade do terreno e assim oferecer mais informações que possa melhorar os padrões da rega do campo.

Por fim, o capitulo das conclusões refere algumas possíveis implementações futuras que possam melhorar o e-NOVAAS. Implementações futuras como por exemplo uma ligação mais direta ao aasx-package explorer de forma que seja possível gerar as estruturas que o e-NOVAAS utiliza diretamente a partir do interface do aasx-package-explorer. Outra implementação sugerida é a expansão do leque de dados que o e-NOVAAS consegue interpretar, atualmente o e-NOVAAS serve como uma prova de conceito e como tal não distingue todos os tipos de dados presentes nos elementos de indústria 4.0, é importante para o futuro que o gerador seja melhorado para suportar estes dados. Outra melhoria ao gerador é que seja capaz de também gerar o norhtbound interface do e-NOVAAS visto que durante os testes notou-se que o processo mais demoroso era a customização do Northbound interface para cada caso de testes.

Para terminar, o e-NOVAAS é uma implementação de referência de AAS para dispositivos de embedded. Foram necessárias alguma tentativas até chegar á implementação atual e mesmo an implementação atual foram encontrados alguns problemas relacionados com gestão de memória, algo que foi corrigido durante o desenvolvimento. Ainda existem alguns pormenores que podem ser melhorados, mas a iteração atual do e-NOVAAS é uma implementação de referência de AAS em dispositivos embutidos e um passo em frente para facilitar a transformação da indútria para indústria 4.0.

Bibliografia

[1] *12.2 manuals - GNU Project*. URL: https://gcc.gnu.org/onlinedocs/12.2.0/ (ver p. 17).

[2] *API Submodel Swagger Hub*. URL: https://app.swaggerhub.com/apis/Plattform_i40/SubmodelServiceSpecification/V3.0.1_SSP-001 (ver p. 25).

[3] G. Booch. *Object-oriented analysis and design: With applications*. Addison-Wesley Professional, 2007 (ver p. 10).

[4] G. Booch e R. Machsimchuk. *Dynamic Languages and Late Binding* (ver p. 9).

[5] *C++ clock()*. URL: https://www.programiz.com/cpp-programming/library-function/ctime/clock (ver p. 17).

[6] *C++ function overriding - javatpoint*. URL: https://www.javatpoint.com/cpp-function-overriding (ver p. 14).

[7] *Can semantic ID(s) be used without defining a concept description within the AAS package?* URL: https://github.com/admin-shell-io/questions-and-answers#id41 (ver p. 28).

[8] *cJSON - GitHub*. URL: https://github.com/DaveGamble/cJSON/tree/master (ver p. 33).

[9] B. J. Cox. *Object-oriented programming an evolutionary approach*. Addison-Wesley, 1987 (ver p. 11).

[10] *Details of the asset administration shell - part 1*. URL: https://www.plattform-i40.de/IP/Redaktion/EN/Downloads/Publikation/Details_of_the_Asset_Administration_Shell_Part1_V3.html (ver pp. 2, 3).

[11] *FAAAST - About the Project*. URL: https://faaast-service.readthedocs.io/en/v0.2.0/about/about/ (ver p. 22).

[12] *FAAAST - Architecture*. URL: https://faaast-service.readthedocs.io/en/v0.2.0/architecture/architecture/ (ver p. 21).

[13] *FAAAST - Asset Connections*. URL: https://faaast-service.readthedocs.io/en/v0.2.0/assetconnections/assetconnection/ (ver p. 21).

[14] *FAAAST - Getting Started*. URL: https://faaast-service.readthedocs.io/en/v0.2.0/gettingstarted/gettingstarted/ (ver p. 21).

[15] *FAAAST - HTTP Endpoint*. URL: https://faaast-service.readthedocs.io/en/v0.2.0/endpoints/http_endpoint/ (ver p. 21).

[16] *FAAAST - Tools for Digital Twins based on Industrie 4.0 standards (Asset Administration Shell)*. URL: https://www.iosb.fraunhofer.de/en/projects-and-products/faaast-tools-digital-twins-asset-administration-shell-industrie40.html (ver p. 21).

[17] *Fraunhofer Faast Documentation*. URL: https://faaast-service.readthedocs.io/en/latest/ (ver pp. 4, 21).

[18] *Fraunhofer IESE*. URL: https://eclipse.dev/basyx/about/ (ver pp. 4, 20).

[19] *Function overloading*. 2022-08. URL: https://en.wikipedia.org/wiki/Function_overloading (ver p. 9).

[20] *Glossary*. URL: https://www.plattform-i40.de/IP/Navigation/EN/Industrie40/Glossary/glossary.html (ver p. 1).

[21] *IDTA Submodels*. URL: https://industrialdigitaltwin.org/en/content-hub/submodels (ver p. 45).

[22] *ISO/IEC/IEEE 24765:2017*. 2022-12. URL: https://www.iso.org/standard/71952.html (ver p. 8).

[23] *Late binding*. 2022-08. URL: https://en.wikipedia.org/wiki/Late_binding#Late_binding_in_C++ (ver p. 10).

[24] *Lesson: Object-oriented programming concepts*. URL: https://docs.oracle.com/javase/tutorial/java/concepts/index.html (ver p. 12).

[25] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (ver p. ii).

[26] *Method overloading in Java*. 2022-09. URL: https://www.geeksforgeeks.org/method-overloading-in-java/ (ver p. 14).

[27] Microsoft. *Use early binding and late binding in automation - office*. 2021-10. URL: https://learn.microsoft.com/en-gb/previous-versions/office/troubleshoot/office-developer/binding-type-available-to-automation-clients (ver p. 8).

[28] G. D. Orio. *README.md · main · NOVAAS collection / NOVAAS Catalog / Nova School of Science and Technology / NOVAAS hydraulic unit simulation · GITLAB*. URL: https://gitlab.com/novaas/catalog/nova-school-of-science-and-technology/novaas-hydraulic-unit-simulation/-/blob/main/README.md (ver pp. 4, 19).

[29] *Polymorphism (computer science)*. 2023-01. URL: https://en.wikipedia.org/wiki/Polymorphism_(computer_science) (ver p. 9).

[30] T. W. PRATT e M. V. ZELKOWITZ. *Programming languages: Design and implementation*. Prentice-Hall, 1999 (ver pp. 9, 11).

[31] SAP-archive. *SAP-Archive/I40-AAS: Provide a set of tools to realize the asset administration shell for Industrie 4.0.* URL: https://github.com/SAP-archive/i40-aas (ver pp. 4, 20).

[32] D. K. Schweichhart. URL: https://ec.europa.eu/futurium/en/system/files/ged/a2-schweichhart-reference_architectural_model_industrie_4.0_rami_4.0.pdf (ver pp. 1, 24).

[33] *serde-json - Docs*. URL: https://docs.rs/serde-json-core/latest/serde_json_core/ (ver p. 31).

[34] H. Singh. *Early binding and late binding in C++*. 2018-02. URL: https://www.geeksforgeeks.org/early-binding-late-binding-c/ (ver p. 11).

[35] *Specification AAS Part 2 API*. URL: https://industrialdigitaltwin.org/en/wp-content/uploads/sites/2/2023/06/V3.0-Specification-AAS-Part-2-API.pdf (ver pp. 21, 25).

[36] S. Swart. *Eclipse basyx™*. 2022-10. URL: https://projects.eclipse.org/projects/dt.basyx (ver pp. 4, 20).

[37] M. Tausig. *Constrained Devices*. URL: https://www.netidee.at/comatrix/constrained-devices (ver p. 4).

[38] *The digital twin in Industrie 4.0 - a short introduction to properties, Submodels amp; Asset Administration shells (AAS)*. 2021. URL: https://www.plattform-i40.de/IP/Redaktion/EN/Downloads/Publikation/Hardshell-softcore_ppt.html (ver p. 3).

[39] *Virtual function in C++*. 2023-01. URL: https://www.geeksforgeeks.org/virtual-function-cpp/ (ver p. 13).

[40] *What is early binding? - definition from Techopedia*. URL: https://www.techopedia.com/definition/25612/early-binding-c (ver p. 8).

[41] *What is Industrie 4.0?* URL: https://www.plattform-i40.de/IP/Navigation/EN/Industrie40/WhatIsIndustrie40/what-is-industrie40.html (ver p. 1).