Ricardo Jorge Ferreira de Sousa Alves

Licenciado em Engenharia Informática

Família de DSLs para *Integrated Modular Avionics*

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador : Doutor Vasco Miguel Moreira Amaral,
Professor Auxiliar, Universidade Nova de Lisboa

Co-orientador : João Miguel Cintra de Jesus Silva,
Gestor de Projetos, GMVIS Skysoft, S. A.

Júri:

Presidente: Doutor António Maria Lobo César Alarcão Ravara

Arguente: Doutor João Carlos Pascoal de Faria

Vogal: Doutor Vasco Miguel Moreira do Amaral

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Setembro, 2014

Família de DSLs para
Integrated Modular Avionics

Para a minha família.

Agradecimentos

Em primeiro lugar, quero expressar o meu agradecimento ao Professor Vasco Amaral, pela disponibilidade demonstrada ao longo da elaboração da dissertação, assim como por todas as oportunidades que me proporcionou.

Agradeço à GMVIS Skysoft, S. A., em particular ao Gestor de Projetos João Silva, por todo o apoio e orientação prestada. Adicionalmente, agradeço a todos os colegas da empresa que tive oportunidade de conhecer e que partilharam comigo conhecimentos chave para realizar esta dissertação.

Aos meus Pais, pessoas que foram sempre um exemplo de coragem, amor e perseverança, por me terem transmitido valores que fizeram de mim a pessoa que sou hoje.

Por último manifesto um sentido e profundo reconhecimento à Cláudia, pelas palavras de incentivo e por todo o apoio incondicional ao longo destes anos.

Resumo

Na atualidade existe a necessidade de produzir novos aviões de forma rápida, eficiente e económica com o objetivo de abrir novas rotas de voo, expansão das já existentes e substituição de aviões em fim de vida útil. Neste contexto, e sem nunca deixar de cumprir os apertados requisitos do domínio que incluem a exigência de elevada qualidade, a indústria adotou a arquitetura IMA que permite executar várias aplicações *aviónicas* num único sistema de computação partilhado.

Com esta arquitetura, o desenvolvimento do *software* para as aeronaves ganhou uma elevada relevância, sendo necessário gerar código automaticamente, reutilizar código já testado proveniente de outras aplicações anteriormente desenvolvidas e garantir o mais cedo possível que o *software* desenvolvido se encontra de acordo com os standards.

Apesar da complexidade do domínio, desenvolveu-se uma ferramenta que permite produzir a estrutura do código de novas aplicações para a *aviónica*. A ferramenta lida com a variabilidade das diversas linhas de produto e reduz o tempo de desenvolvimento.

Uma DSL poderia ser uma solução apropriada, pois permite cobrir diversos requisitos exigidos, no entanto, esta solução não é exequível porque seria necessário produzir uma linguagem para cada configuração de *software aviónico* diferente se pretendêssemos tirar partido da especificidade. Com esta dissertação, solucionou-se esta dificuldade com recurso à noção de família de DSLs. *Trata-se de um conjunto de linguagens para um domínio específico, que apresentam um conjunto comum de conceitos chave, mas que adaptam alguns desses conceitos para cumprir a variabilidade dos requisitos*.

Utilizou-se a abordagem MDD para desenvolver um gerador automático de DSLs que é capaz de produzir a linguagem desejada de acordo com a configuração de *software* pretendida para a partição pertencente a um módulo *aviónico*.

As linguagens geradas apresentam um nível de usabilidade adequada para o domínio, bem como têm a capacidade de validar as construções efetuadas usando a DSL e produzir os artefactos pretendidos.

Palavras-chave: Integrated Modular Avionics, Model-Driven Development, Domain Specific Language, Software Product Line, Família de DSLs, Fábrica de DSLs.

Abstract

Nowadays there is a need to produce new aircrafts quickly, efficiently and economically in order to open new flight routes, expand existing aircrafts and replace end-of-life aircrafts. In this context, never failing to meet the tight requirements of the domain including the requirement of high quality, the industry has adopted the IMA architecture that lets one run multiple avionics applications on a single shared computer system.

With this architecture, the software development for avionic systems gained high relevance, being necessary to generate code automatically, reusing already tested code from other previously developed applications and ensure the earliest possible that software is developed according to the standards.

Despite the high complexity of the domain, we developed a tool that enables to produce the code structure of the new applications for avionics. The tool deals with the variability of the various product lines and reduces development time.

A DSL is an appropriate solution, because it allows cover various requirements, however, this solution is not feasible because it would be necessary to produce a language for each different setting of avionics software if we wished to take advantage of the specificity.

With this dissertation, this difficulty was solved by using the notion of family of DSLs. *This is a set of languages for a given domain, providing a common set of key concepts, but some of these concepts are tailored to meet the requirements of the variability*.

It was used the MDD approach to develop an automatic generator of DSLs which is capable of producing the desired language according the software configuration for an avionic module.

The languages generated have an adequate level of usability for the domain as well as the ability to validate the constructs made using DSL and produce the desired artifacts.

Keywords: Integrated Modular Avionics, Model-Driven Development, Domain Specific Language, Software Product Line, Family of DSLs, Factory of DSLs.

Lista de Abreviaturas

ANT Another Neat Tool

APEX APplication EXecutive

API Application Programming Interface

ARINC Aeronautical Radio INCorporated

DOM Document Object Model

DSL Domain Specific Language

ECL Epsilon Comparison Language

EGL Epsilon Generation Language

EMF Eclipse Modeling Framework

EML Epsilon Merging Language

EOL Epsilon Object Language

ESA European Space Agency

ETL Epsilon Transformation Language

EVL Epsilon Validation Language

FAQ Frequently Asked Question

FIFO First In, First Out

FMS Flight Management System

FODA Feature-Oriented Domain Analysis

FPV Flight Path Viewer

GME Generic Modeling Environment

GMF Graphical Modeling Framework

GOPPRR Graph Object Property Port Role Relationship

GPL General Purpose Language

GReAT Graph Rewriting And Transformation

HM Health Monitor

HOT Higher-Order Transformation

IDE Integrated Development Environment

IMA Integrated Modular Avionics

IMADE Integrated Modular Avionics Development Environment

IMA-SP Integrated Modular Avionics for SPace

JS JavaScript

LMW Languages Metamodeling Workbench

MDD Model-Driven Development

NGMP Next Generation MicroProcessor

OCL Object Constraint Language

OMG Object Management Group

PMK Partition Management Kernel

QVT Query/View/Transformation

RCP Rich Client Platform

SPL Software Product Line

TTM Time To Market

UAV Unmanned Aerial Vehicle

UML Unified Modeling Language

XML eXtensible Markup Language

Conteúdo

Lista de Figuras

Lista de Tabelas

1

Introdução

1.1 Introdução Geral

Na atualidade presencia-se um processo contínuo de aprofundamento da integração económica, social, cultural e política a nível mundial. A este processo chamamos de globalização, e só é possível devido à massificação dos transportes e comunicações.

A aviação tem um papel central neste processo, pois permite transportar pessoas e bens de forma rápida dentro de um país, bem como entre continentes, sem necessitar de infraestruturas físicas que interliguem continuamente a origem ao destino.

A natureza particular do meio de transporte aéreo, não permite paragens em pleno voo para manutenção e reparação, ao contrário de transportes terrestres e marítimos. Esta limitação origina que os aviões fiquem sujeitos a fortes requisitos de construção e operação, como é exemplo a segurança, performance, integridade, confiança e tolerância a falhas como referido em [21].

Devido à necessidade de serem garantidos os requisitos de construção e operação, com o objetivo de permitir o voo das aeronaves no espaço aéreo de diversos países, as autoridades nacionais e internacionais especificaram standards que têm imperativamente de ser cumpridos, culminando num processo de certificação dos aviões, sistemas e componentes que o constituem. Estas limitações rígidas prendem-se com a necessidade de salvaguardar a vida dos passageiros, integridade dos bens transportados, assim como, todo o ambiente envolvente onde os aviões circulam.

Com o aprofundamento da globalização existe consequentemente um maior número de passageiros que efetuam viagens aéreas, aumentado a necessidade de produzir novos aviões de uma forma mais rápida, eficiente e económica, não só com a finalidade de abrir novas rotas de voo ou expansão das já existentes, mas também para substituição de aviões em fim de vida útil.

No contexto de aumento de produção e diminuição de custos, sem nunca deixar de

cumprir os requisitos do domínio, a indústria aeronáutica adotou do meio militar a arquitetura *Integrated Modular Avionics* (IMA). Esta arquitetura tem como objetivo suportar a execução de múltiplas aplicações com funções de *aviónica*[1] num único sistema de computação partilhado, ao invés das arquiteturas federadas que utilizam *hardware* independente para cada aplicação.

A arquitetura IMA permite diminuir os custos de construção, manutenção e operação, pois é reduzida a necessidade de um elevado número de cabos e sistemas de computação a bordo. Na atualidade verifica-se que já existem alguns aviões civis de última geração a utilizarem esta arquitetura, e o caminho que se encontra a ser percorrido pela indústria baseia-se na utilização massiva da mesma.

O desenvolvimento de aplicações para a aeronáutica sempre careceu da necessidade de cumprir os standards de forma rigorosa, no entanto, com esta nova arquitetura e a necessidade de construção de um elevado número de aviões, colocam-se novos desafios ao desenvolvimento de *software*. As linguagens de domínio específico são vistas como parte da solução, a fim de diminuir os custos de desenvolvimento e de certificação, facilitando também a entrada de novas empresas neste nicho de mercado.

1.2 Motivação

Tratando-se o *software* de um fator importante e indispensável para o funcionamento dos equipamentos de bordo de um avião moderno, com a importação da arquitetura IMA, a importância do desenvolvimento do *software* para *aviónica* ganhou relevância.

Atualmente, os engenheiros informáticos têm necessidade de desenvolver mais e melhor. Uma das formas de responder a esta necessidade é recorrer a mecanismos de geração automática do código, que têm por base a reutilização de *software* planeado ou oportunista [13].

Outra forma de melhorar a produção de *software* é promover a reutilização parcial de código repetitivo e já testado proveniente de aplicações que já foram desenvolvidas para outros aviões, pois atualmente esta reutilização é reduzida.

Com os dois princípios referidos anteriormente, é possível garantir que existem menos erros de implementação, logo não é necessário despender tempo das equipas de desenvolvimento na sua correção. Esta situação origina uma diminuição do custo final do *software*.

Com estas técnicas, é também possível concluir um produto em menos tempo e consequentemente melhorar o *Time To Market* (TTM), pois entrega-se mais cedo ao construtor as aplicações para o seu avião.

Uma empresa que desenvolva aplicações para este domínio, tem de garantir que as mesmas se encontram de acordo com os standards, sendo necessário certificá-las através das autoridades competentes. A certificação é dispendiosa e para que a mesma possa

[1] Termo utilizado para fazer referência à eletrónica das aeronaves (provém de AVIação + eletrÓNICA).

ter sucesso, é necessária a entrega de diversos documentos relacionados com o *software* desenvolvido e aguardar vários meses.

O processo de certificação pode ser acelerado caso sejam utilizadas no desenvolvimento ferramentas qualificadas pelas autoridades certificadoras e que possam produzir alguns artefactos necessários à certificação.

1.3 Descrição do Problema

A aviação é um domínio complexo que apresenta vários níveis no processo de desenvolvimento *aviónico* das aeronaves, sendo assim esta dissertação foca-se no nível zero da figura 1.1. A aviação é regulada por entidades exigentes que impõem um elevado número de standards que têm obrigatoriamente de ser cumpridos. Desta exigência surge a necessidade de produzir o mais cedo possível *software* que já se encontra de acordo com os mesmos. Assim podem-se reduzir custos com o processo de certificação.

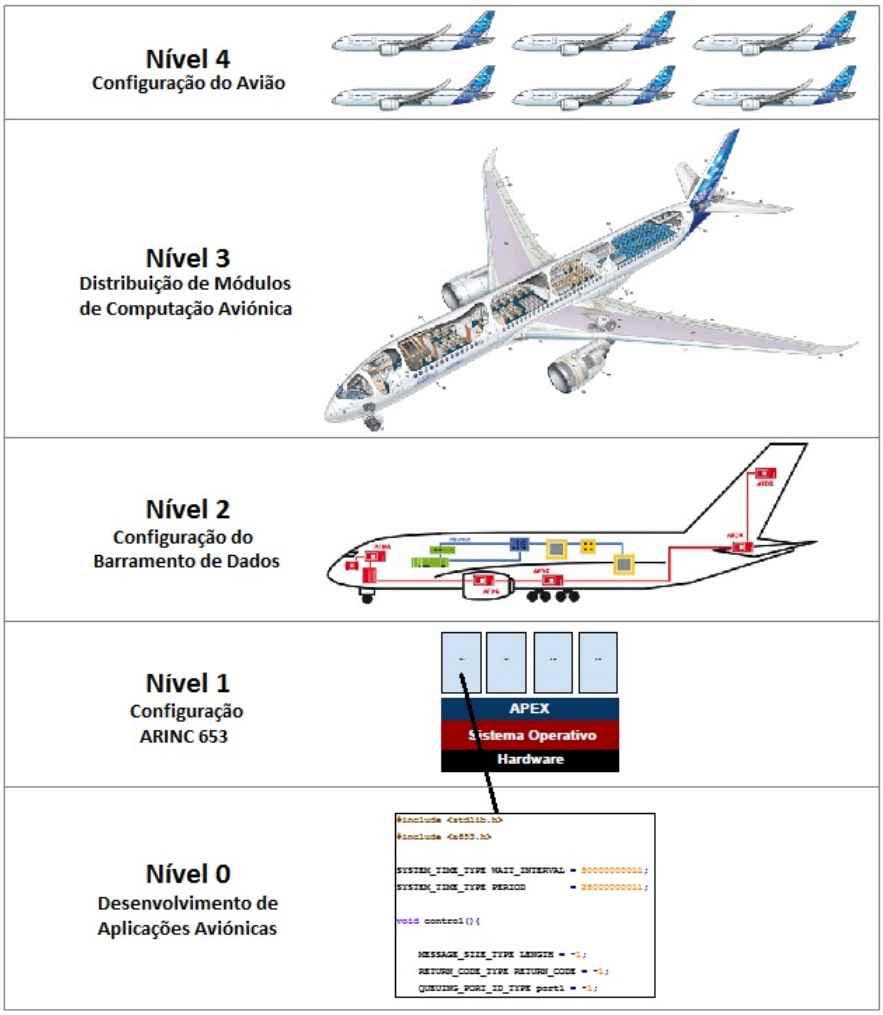


Figura 1.1: Processo de desenvolvimento *aviónico* das aeronaves

Outro dever, é a necessidade de efetuar a validação e verificação do código produzido,

bem como a produção de artefactos necessários para a certificação do diverso *software* desenvolvido. Este dever prende-se com o objetivo de passar pelo processo de certificação o menor número de vezes possível, diminuindo de forma consequente os custos de produção.

A reutilização de módulos de código certificados é algo que na atualidade é efetuado de forma reduzida, no entanto, caso seja efetuado com frequência, facilita-se a certificação de novo *software*.

Adicionalmente às exigências anteriores, e não menos importante, é a necessidade de acelerar o desenvolvimento de *software* para aeronaves com características diferentes.

Em conjunto, todos os requisitos referidos, culminam num problema complexo de engenharia que necessita de uma abordagem metódica e científica para produzir uma solução.

O facto de existirem diversas aeronaves com características diferentes faz transparecer que cada avião tem uma configuração de *software* diferente, no entanto, o que existem são diversas linhas de produto, ou seja, diversos aviões que partilham uma determinada configuração de *software*.

Nesta dissertação desenvolveram-se ferramentas que permitem efetuar o desenvolvimento da estrutura do código de novas aplicações para *aviónica* no contexto IMA, produzindo também alguns artefactos necessários para a sua certificação. Estas ferramentas lidam principalmente com a variabilidade das diversas linhas de produto e reduzem o tempo de desenvolvimento.

A produção de uma *Domain Specific Language* (DSL) adequa-se às necessidades apresentadas, no entanto existiu um desafio de elevada complexidade, pois foi necessário investigar como desenvolver uma DSL para IMA tendo em conta as características específicas do domínio.

1.4 Contexto

Esta dissertação realizou-se em contexto empresarial na GMVIS Skysoft, SA. Trata-se de uma empresa de engenharia especializada no desenvolvimento de *software* para as indústrias aeroespacial, aeronáutica, defesa e outras.

Com a adoção em larga escala da arquitetura IMA pelos construtores de aviões, a empresa pretendeu colocar-se numa posição de vantagem face à concorrência, produzindo aplicações para a aeronáutica de forma eficiente e de elevada qualidade. Em complemento, é objetivo da empresa produzir um ambiente de desenvolvimento integrado para a arquitetura IMA. Nesse sentido, o protótipo desta dissertação mostrou-se uma peça fundamental e foi integrado como parte desse pacote de *software* ainda em desenvolvimento. Atualmente o protótipo é utilizado internamente, mas poderá vir a ser comercializado.

A empresa adquiriu e cimentou conhecimentos na área de modelação e produção de

linguagens para domínios específicos, melhorando desta forma os seus processos internos de desenvolvimento de *software* de qualidade.

1.5 Solução

Uma solução para o problema exposto na secção 1.3, careceu de um ambiente fácil de utilizar, recorrendo-se a uma interface gráfica intuitiva, que posteriormente permite gerar automaticamente os artefactos desejados.

A utilização de uma DSL pareceu a solução apropriada, pois abrangem-se diversas necessidades, tal como, o aumento da produtividade, pois permite mais rapidamente um especialista entender e desenvolver o que pretende, e possibilita uma melhor comunicação entre os implementadores e os especialistas do domínio, como referido em [38].

No ponto de vista operacional trata-se de uma mais valia, pois uma DSL pode ser gráfica criando um ambiente de operação intuitivo. Outro aspeto importante é a sua construção poder ser efetuada com recurso a modelos [47], o que permite fazer com grande facilidade verificações e validações, bem como criar novas transformações a esses modelos para gerar novos artefactos futuramente pretendidos.

De acordo com as vantagens apresentadas, o desenvolvimento de uma *Domain Specific Language* para cada configuração de *software*, permite a produção mais rápida de todos os artefactos necessários para a certificação, bem como a estrutura do *software*. Esta solução apresentou uma falha central, nomeadamente a necessidade de produzir uma linguagem nova sempre que surgisse uma configuração diferente, encarecendo o processo de desenvolvimento da aeronave e contrariando o que se pretendia.

A solução anterior não foi completamente desprezada, observando-se a mesma, de outro prisma. Assumiu-se que se trata de uma família de linguagens que partilham uma estrutura comum, ao invés de diversas linguagens distintas.

Com esta solução de alto nível apresentada, deparou-se com a necessidade de criar uma *Software Product Line* (SPL) de DSL, por outras palavras, uma linha de linguagens para domínio específico. Trata-se de um conceito inovador que foi explorado na presente dissertação, que visa garantir a geração de um conjunto de linguagens ligeiramente diferentes para cada configuração de *software*, de forma automatizada e simultaneamente garantir todas as restrições do domínio já anteriormente referidas.

Uma SPL de DSL é uma família de linguagens que cresce quando é necessário criar uma nova linguagem mantendo diversas características comuns, existindo para isso um gerador que é capaz de manipular um metamodelo que descreve o domínio. A manipulação permite adicionar, remover e alterar elementos do metamodelo, ou seja, através de uma transformação [30, 32] gera-se uma especificação do metamodelo inicial, e como consequência produz-se uma nova linguagem da mesma família.

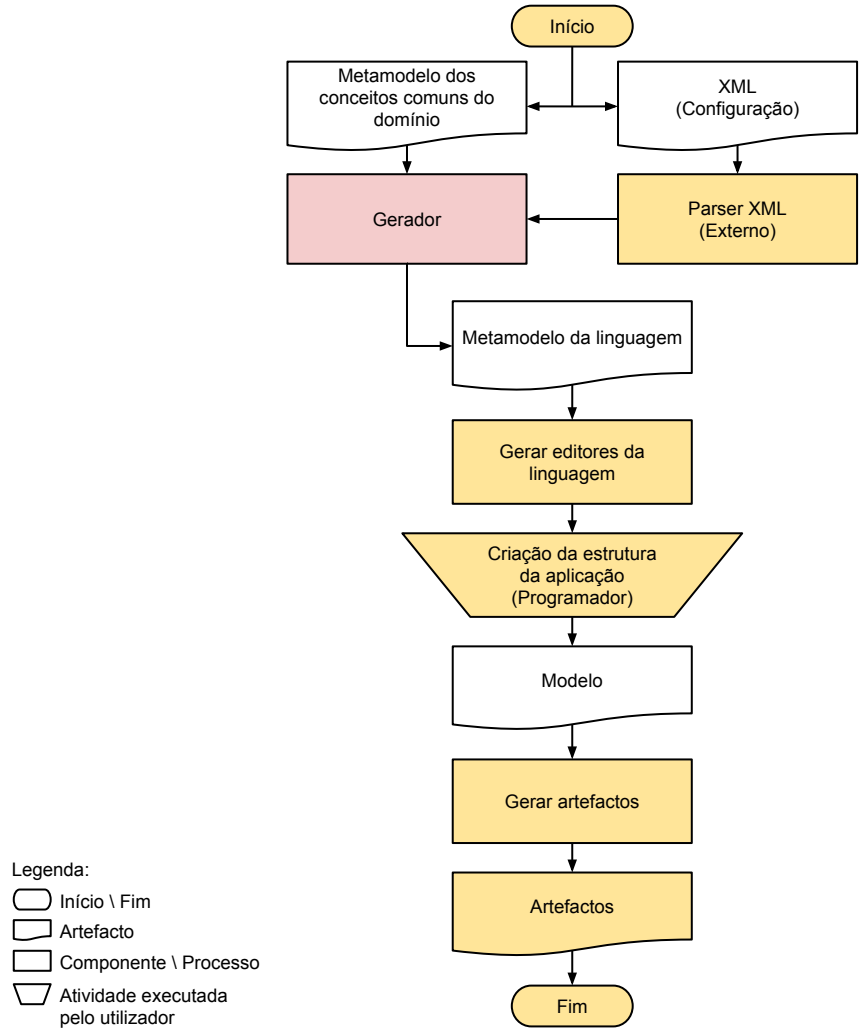


Figura 1.2: Processo genérico da solução

Na figura 1.2 apresenta-se o processo da solução implementada considerando-se o contexto de utilização. No topo à esquerda encontra-se o metamodelo dos conceitos comuns do domínio que descreve as características obrigatórias que todas as linguagens têm de apresentar. Posteriormente, através de uma transformação de composição, compõe-se o metamodelo anterior com o metamodelo dos conceitos de configuração. Este último é produzido no interior do gerador através das informações que se obtém do *Parser* XML.

Neste processo surgiram diversas barreiras que tiveram de ser transpostas, nomeadamente o desenvolvimento de um *Parser* XML otimizado, e a escolha correta da API que é disponibilizada ao gerador responsável por produzir o metamodelo da linguagem e os restantes artefactos.

A maior dificuldade que necessitou de uma profunda investigação consistiu no processo para gerar o metamodelo da linguagem, ou seja, determinar qual o modo apropriado para gerar de forma correta o metamodelo da DSL.

Uma vez que este processo decorre em tempo de execução, ou seja, quando o utilizador se encontra à espera, então o fator de eficiência ganhou relevância, bem como garantir que se produz uma linguagem apropriada para o domínio.

Depois do metamodelo da linguagem e dos restantes artefactos estarem criados, é então gerado o editor gráfico da linguagem, operação que é efetuada com a ferramenta adequada (ver na secção 3.1.1.3). As etapas seguintes relacionam-se com a utilização da DSL produzida.

O gerador é o elemento principal deste processo, pois é responsável por gerar não só o metamodelo da linguagem, mas também todos os restantes artefactos necessários para produzir uma DSL da família de DSLs.

Do ponto de vista do utilizador que pretende desenvolver uma aplicação *aviónica* para executar numa dada partição, necessita em primeiro lugar de iniciar a execução do gerador de DSLs, selecionar o ficheiro XML que contém a configuração do módulo, bem como a partição que irá executar essa aplicação. Posteriormente o gerador produz a DSL que poderá ser finalmente utilizada. O gerador de DSLs foi batizado de *IMA Studio*.

1.6 Estrutura da Dissertação

Este documento encontra-se organizado da seguinte forma:

- **Capítulo 1 - Introdução:** Apresenta-se uma visão geral da dissertação;
- **Capítulo 2 - Estado da Arte:** Documenta-se a informação concretizada nas áreas mais importantes que contribuíram para o desenvolvimento da presente dissertação;
- **Capítulo 3 - Tecnologias Relacionadas:** Expõe-se um estudo comparativo de ferramentas que foram essenciais para o desenvolvimento do protótipo;
- **Capítulo 4 - Análise do Domínio:** Apresenta-se o estudo e a informação importante sobre o domínio, nomeadamente IMA;
- **Capítulo 5 - Solução:** Descreve-se do ponto de vista teórico a solução para o problema apresentado na dissertação;
- **Capítulo 6 - Implementação:** Apresentam-se os detalhes de implementação relativos ao protótipo desenvolvido;
- **Capítulo 7 - Avaliação de Usabilidade:** Expõe-se a sequência de etapas que foram realizadas na execução da avaliação, bem como os resultados obtidos;
- **Capítulo 8 - Conclusões:** Resumem-se as considerações mais importantes que se apresentaram no final de diversos capítulos deste documento;
- **Capítulo A - Anexos:** Encontram-se diversos elementos, que devido à sua relevância foram incorporados na presente dissertação.

2

Estado da Arte

2.1 *Integrated Modular Avionics* (IMA)

O IMA é uma arquitetura dos sistemas de eletrónica e computação de uma aeronave, onde o seu *hardware* e *software* de propósito geral se encontram descritos nos standards ARINC 650 [8] e ARINC 651 [9]. Numa perspetiva concreta, o IMA é uma rede de módulos físicos de computação em tempo real.

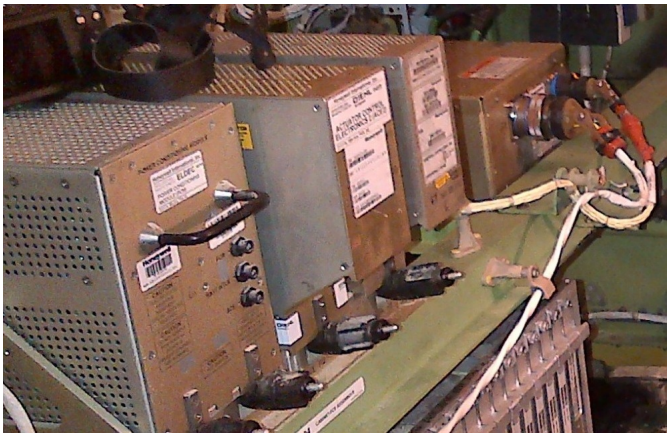


Figura 2.1: Módulos físicos de computação na baía *aviónica*

Os módulos físicos de computação (figura 2.1) suportam a execução de múltiplas aplicações com funções de *aviónica* num único sistema de computação partilhado, ou seja, permitem que um conjunto de aplicações possam ser corretamente executadas no mesmo *hardware*. Cada aplicação apresenta um comportamento exterior e desempenha as suas funções tal como se estivesse a ser executada num sistema de computação dedicado.

O desenvolvimento de aplicações com funções de *aviónica* é simplificado, pois existe uma API comum para aceder aos recursos do *hardware*. A existência desta API também permite que os implementadores se foquem no desenvolvimento das aplicações, evitando o risco de efetuarem erros em camadas de *software* de níveis inferiores.

A arquitetura IMA permite aumentar a disponibilidade das funções *aviónicas* de uma aeronave, pois em caso de falha num determinado módulo físico principal é possível reconfigurar as aplicações para serem executadas noutro módulo físico sobresselente.

As comunicações entre módulos físicos de computação são efetuadas com recurso a redes externas partilhadas que cumprem os standards ARINC 429 [7] ou ARINC 664 [12].

A maioria das aeronaves que se encontram em operação, têm uma arquitetura que se caracteriza pela utilização de sistemas federados, ou seja, possuem uma arquitetura que é proprietária de cada fabricante e requerem *hardware* específico de computação segregado fisicamente para cada função de *aviónica*. Adicionalmente esta arquitetura exige cabos de interligação dedicados entre cada um dos computadores.

Nas aeronaves de última geração, como é exemplo o Airbus A380 ou o Boing 787 Dreamliner, a *aviónica* tem uma arquitetura IMA que permite reduzir o número de computadores e cabos existentes. Consequentemente diminui-se a complexidade dos sistemas, o peso total dos equipamentos, a quantidade de energia consumida e a certificação é facilitada, ou seja, os custos de produção, manutenção e operação das aeronaves que utilizam esta arquitetura reduzem.

A Airbus[1] e a Boing[2] reportaram poupanças de 25%, 50% e 60% em peso, energia e volume, respetivamente [24, 18].

Esta arquitetura também é utilizada noutros domínios além da aviação civil e militar, como é exemplo nos *Unmanned Aerial Vehicles* (UAV) ou Satélites.

2.1.1 Standard ARINC 653

Em foco, na presente dissertação encontra-se o standard ARINC 653 [10, 11]. Este standard especifica a API da arquitetura IMA entre o sistema operativo de um módulo *aviónico* e a aplicação *aviónica*. Outro aspeto também especificado é o comportamento do sistema operativo do módulo *aviónico*, de forma a garantir a partição de espaço e tempo para execução de diversas aplicações no mesmo módulo de computação.

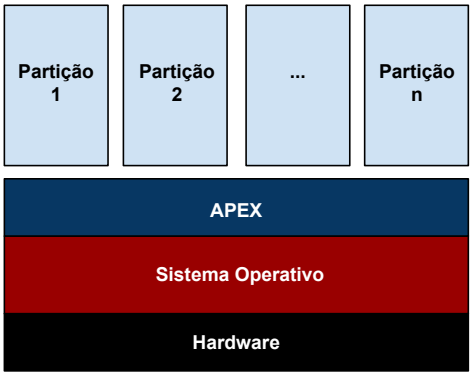


Figura 2.2: Arquitetura do sistema dentro do módulo de computação

[1] http://www.airbus.com/
[2] http://www.boeing.com/

A figura 2.2 demonstra a arquitetura do sistema que se encontra dentro de um módulo de computação IMA. Sobre o *hardware* do módulo encontra-se instalado um sistema operativo de tempo real com características específicas de forma a conseguir fornecer o comportamento definido no standard ARINC 653. O sistema operativo fornece serviços às partições onde se encontram instaladas as aplicações *aviónicas*, através de uma API bem definida denominada de *APplication EXecutive* (APEX).

Quando a indústria aeronáutica pretende desenvolver um módulo IMA, é definido um conjunto de configurações que impõem algumas limitações às partições, afetando diretamente as aplicações *aviónicas* instaladas em cada partição. Estas configurações são guardadas num ficheiro XML que é essencial para iniciar o sistema operativo do módulo.

Na presente dissertação utiliza-se este ficheiro XML para produzir uma família de DSLs com o objetivo de permitir efetuar o desenvolvimento da estrutura do código de novas aplicações de *aviónica* tendo em conta as caraterísticas do módulo e da partição alvo.

2.1.2 Processo de Desenvolvimento de *Software*

Na aeronáutica, utiliza-se habitualmente o processo de desenvolvimento de *software* designado por modelo V. Este considera-se uma extensão do modelo em cascata, e demonstra a relação entre cada fase do ciclo de vida de desenvolvimento e a sua fase de testes associada.

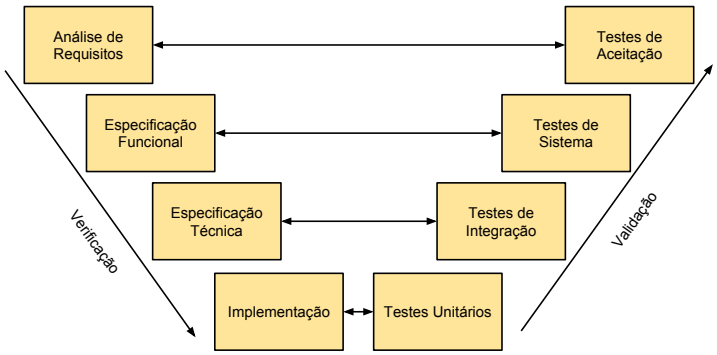


Figura 2.3: Modelo V

Na figura 2.3 apresenta-se o modelo V [27], o eixo horizontal e vertical representa o tempo e o nível de abstração, respetivamente.

No final de cada etapa de desenvolvimento é realizada uma avaliação, verificação e validação dos produtos gerados. Desta forma a deteção de erros ocorre o mais cedo possível com intuito de reduzir os custos da sua correção.

2.1.3 Certificação

No desenvolvimento de qualquer componente de *hardware* ou *software* que integre uma aeronave é necessária certificação que confirme o cumprimento dos standards que

lhe são exigidos. No âmbito do ARINC 653 [10, 11], o principal documento que serve de guia para o desenvolvimento de *software* para os sistemas de bordo é o DO-178C [21].

As autoridades responsáveis não certificam *software* separado do *hardware*, ou seja, é necessário certificar o *software* quando este está instalado no equipamento. No entanto, caso exista reutilização de componentes de *software* que já foram certificados, isto para desenvolver novas aplicações para outro *hardware*, então o processo é agilizado. Esta situação também se verifica com a utilização de ferramentas qualificadas no ciclo de vida de desenvolvimento das aplicações.

Alguns dos artefactos necessários para a certificação são o código fonte, testes, configurações, relatórios referentes ao desenvolvimento, entre outros.

2.1.4 Sistemas Operativos IMA

2.1.4.1 AIR

O AIR[3] nasceu do interesse da ESA[4] pelos princípios do IMA e ARINC 653. Este sistema operativo foi desenvolvido com o objetivo de provar conceitos e demonstrar o seu uso no espaço. Posteriormente, uma iniciativa industrial continuou o trabalho efetuado até então, com o objetivo de o tornar um sistema operativo para utilização real [25].

Este sistema operativo implementa o APEX definido no ARINC 653 e a API definida no *Integrated Modular Avionics for SPace* (IMA-SP), utilizando tecnologia de virtualização. Cada partição tem o seu sistema operativo independente e são virtualizados e executados no topo do *Partition Management Kernel* (PMK).

O PMK é o núcleo do sistema operativo que lida com o *scheduling* e *dispatching* das partições, bem como, com as comunicações entre partições.

2.1.4.2 PikeOS

O PikeOS[5] é um sistema operativo de tempo real para sistemas embebidos. É utilizado em diversos domínios como o espaço, defesa, medicina e outros.

Trata-se de um sistema operativo com diversas certificações, nomeadamente na aviação, onde a sua utilização é efetuada em aviões de última geração como o Airbus A350. Trata-se de uma plataforma de virtualização que permite executar diversas aplicações que podem ou não ser de tempo real.

No caso em que as aplicações têm requisitos de tempo real, o PikeOS garante que a execução é determinística em termos de espaço e de tempo.

[3] http://air.di.fc.ul.pt/ e http://www.gmv.com/en/aeronautics/products/air/
[4] http://www.esa.int/
[5] http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/

2.1.4.3 VxWorks 653

O VxWorks 653[6], de forma idêntica aos anteriores, trata-se de um sistema operativo de tempo real que permite a execução de aplicações com diferentes níveis de criticidade da segurança.

Este sistema operativo, beneficia da experiência da empresa adquirida no desenvolvimento do VxWorks que é utilizado em sistemas federados.

Atualmente o VxWorks 653 é utilizado no Boing 787 Dreamliner, entre outros.

2.1.5 DSLs para IMA

2.1.5.1 ConfiguIMA

O ConfiguIMA (figura 2.4) foi desenvolvido utilizando uma abordagem orientada pelo código. Trata-se de uma linguagem que permite produzir o ficheiro XML que contém as configurações de um módulo *aviónico* de acordo com o standard ARINC 653. Este ficheiro é crucial para o módulo funcionar, bem como o seu sistema operativo.

Neste caso em particular, esta ferramenta foi desenvolvida com intuito de configurar o sistema operativo AIR (ver na secção 2.1.4.1), no entanto visto o ficheiro produzido cumprir o standard, então este é independente do sistema operativo e funciona em qualquer modulo *aviónico*.

Algumas das configurações mais importantes são o número, tipo, escalonamento e quantidade de memória das partições, o número e tipo de portos de comunicação entre partições, tabelas de erros e respetivas ações, entre outros.

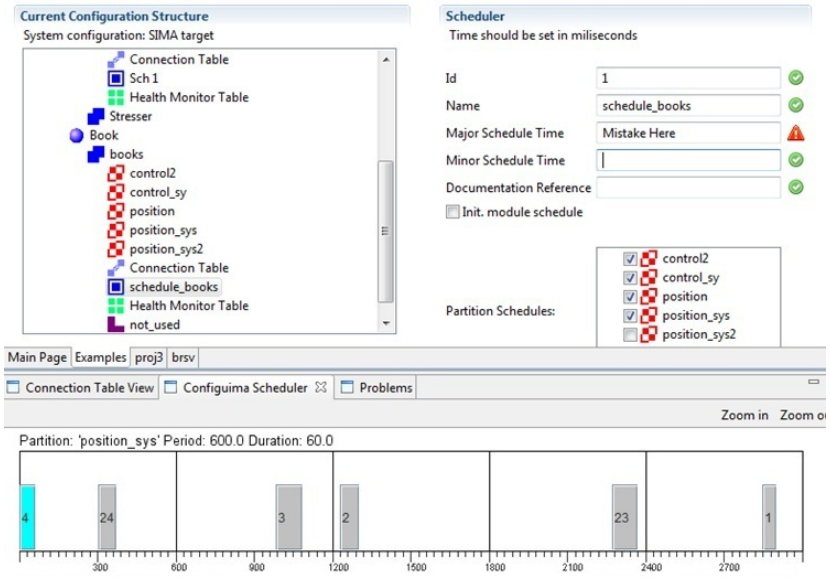


Figura 2.4: Interface gráfica do ConfiguIMA

Esta ferramenta armazena as configurações num servidor MySQL, permitindo que

[6] http://www.windriver.com/products/platforms/safety_critical_arinc_653/

múltiplos utilizadores trabalhem de forma distribuída. Os erros são verificados em tempo de execução e os utilizadores são alertados dos problemas encontrados.

A aplicação produz os ficheiros XML que são utilizados para gerar a família de DSLs com recurso ao protótipo desenvolvido no âmbito da presente dissertação.

2.1.5.2 MIMAD

O MIMAD (figura 2.5) permite desenhar uma aplicação IMA. Esta DSL utiliza a linguagem SIGNAL [42] para efetuar a descrição, refinamento e verificação formal da aplicação.

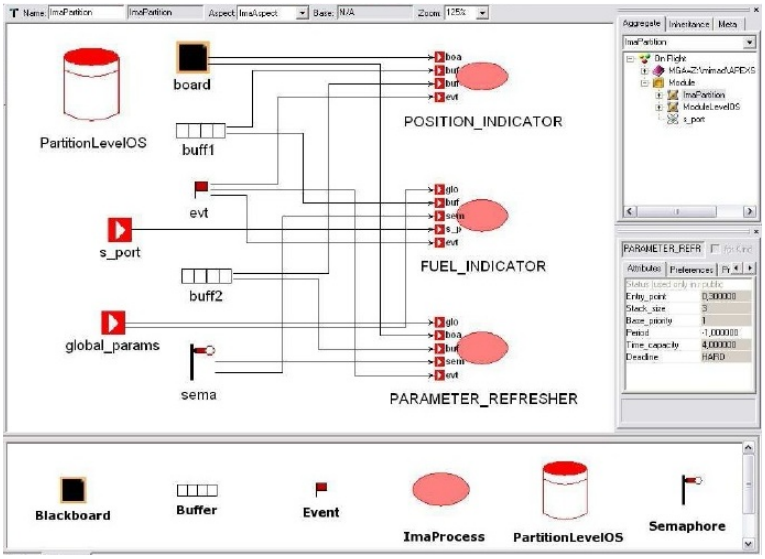


Figura 2.5: Interface gráfica do MIMAD

De acordo com o relatório técnico [1], esta DSL permite criar o código de cada partição, provavelmente utilizando transformações de modelos para texto, escolhendo a linguagem alvo de propósito geral que se pretende obter. Não foi possível obter informação se essas transformações se encontram implementadas.

Confrontando o MIMAD com o protótipo desenvolvido na presente dissertação, verifica-se que o protótipo utiliza a noção de família de DSLs tal não se verifica com o MIMAD, sendo por isso um elemento inovador neste contexto.

2.1.5.3 SCADE Suite

O SCADE Suite[7] (figura 2.6) é um ambiente integrado de desenvolvimento baseado em modelos para *software* crítico embebido. Apresenta uma linguagem baseada em *state flow* e *data flow*. Permite efetuar o desenho, simulação, verificação e geração de código da aplicação.

[7] http://www.esterel-technologies.com/products/scade-suite/

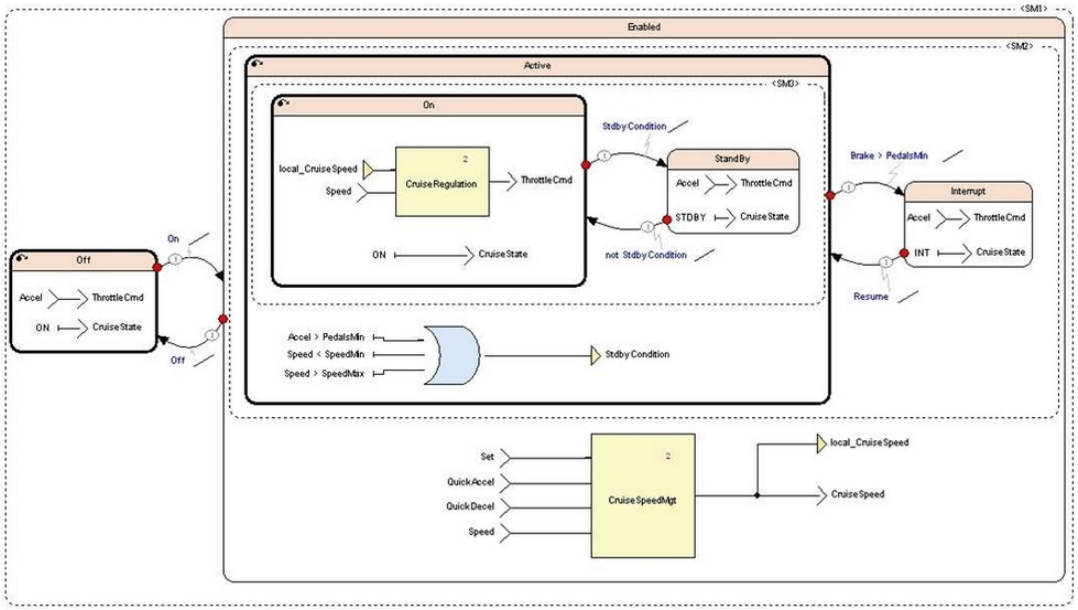


Figura 2.6: Interface gráfica do SCADE Suite

A aplicação vem acompanhada de uma API para o *Eclipse Modeling Framework* (EMF), o que promove a possibilidade do engenheiro informático desenvolver código usando ferramentas externas.

Não foi possível obter informação aprofundada sobre este conjunto de ferramentas, pois existe um custo elevado para obtenção da licença de utilização. Neste sentido, foi efetuada uma análise com recurso à informação e documentação livre.

Da análise verificou-se que a ferramenta produz o código completo, o que por vezes se torna indesejado pelos programadores, pois existe um grande fosso entre o nível em que é efetuada a modelação, e o nível que diz respeito à produção do código para aplicação. Quer-se com isto dizer, que o código produzido pode não corresponder exatamente ao que se pretende, nesse caso é preciso rever o código e alterar o mesmo quando necessário. Com esta solução despende-se tempo a entender o código produzido.

O SCADE de forma idêntica ao MIMAD não utiliza a noção de família de DSLs que foi utilizado na presente dissertação. Noutra perspetiva, o protótipo da dissertação permite construir o esqueleto do código da aplicação deixando só a margem de manobra suficiente para os implementadores das aplicações.

2.2 *Model-Driven Development* (MDD)

O MDD é um paradigma de desenvolvimento de *software* que utiliza modelos como artefacto principal do ciclo de vida do processo de desenvolvimento [30, 6, 44].

Neste paradigma todo o conhecimento é explícito e pode ser modelado, desde os requisitos ao código, garantindo a conformidade e consistência entre modelos. A partir de modelos e transformações de modelos é gerado código e outros artefactos pretendidos.

Este processo normalmente é feito automaticamente, tratando-se de uma característica que distingue o MDD de outros paradigmas que utilizam a modelação.

A automatização do processo de desenvolvimento é uma preocupação do MDD, onde os artefactos podem ser derivados a partir da informação constante nos modelos.

Este paradigma apresenta diversas vantagens, tais como, o aumento da produtividade [46] devido à geração automática de código, tratando-se de um processo rápido e que evita a introdução de erros pelo implementador.

A fácil manutenção é outra vantagem de extrema relevância, por exemplo, quando se pretende criar novos artefactos, basta criar uma nova transformação que se aplica aos modelos já existentes, não sendo necessário alterar o projeto completo. Partindo da vantagem anterior, verifica-se que existe uma elevada reutilização, pois o mesmo modelo pode ser utilizado como *input* de diversas transformações diferentes.

A consistência no MDD é garantida pela geração automática, ao contrário de alterações manuais efetuadas ao código, que se podem verificar noutros paradigmas de desenvolvimento.

2.2.1 Modelo e Metamodelo

2.2.1.1 Modelo

Na engenharia de *software*, o modelo é um artefacto desenvolvido com recurso a uma linguagem de modelação, como a *Unified Modeling Language* (UML). Trata-se de um conjunto de diagramas formais que descrevem um determinado sistema [47].

Um modelo é baseado num artefacto original e reflete só as características importantes do mesmo, tendo em conta o seu objetivo. Para determinados propósitos mais específicos, pode ser usado no lugar do artefacto original [47].

Implicitamente, um modelo implica abstração, ou seja, é perdida informação face ao artefacto modelado. Este facto permite que os engenheiros informáticos e os *stakeholders* tratem eficazmente as suas preocupações [30].

2.2.1.2 Metamodelo

Um metamodelo é um modelo de modelos [47]. Trata-se de um artefacto responsável pela definição da linguagem, ou seja, a sua sintaxe abstrata. Esta define a construção da linguagem de acordo com as regras, propriedades e relações.

É possível produzir modelos utilizando uma linguagem, e nesse caso, estão-se a definir instâncias de um modelo bem formado de acordo com o seu metamodelo.

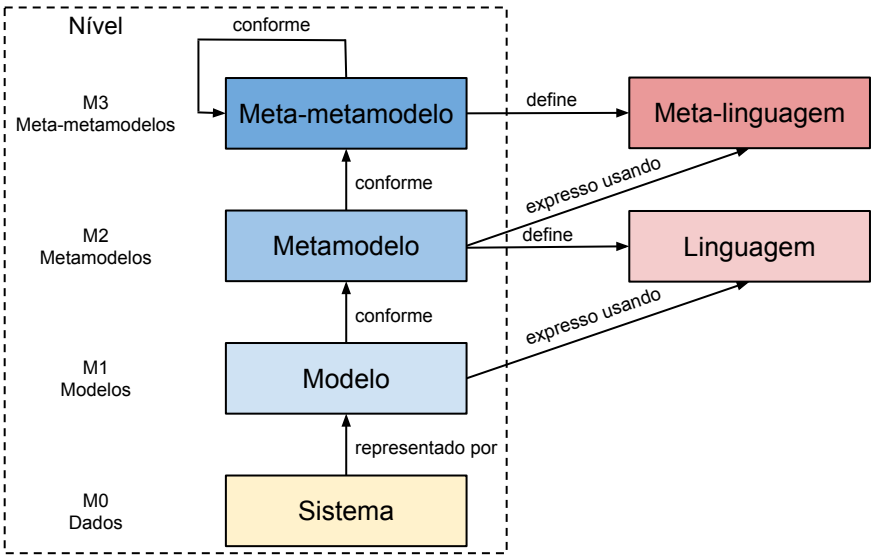


Figura 2.7: Arquitetura de quatro níveis e linguagens de definição (adaptado de [47, 32])

Na figura 2.7 demonstra-se a arquitetura definida pela OMG[8]. Esta é composta pelos níveis dos dados, modelos, metamodelos e meta-metamodelos. No fundo à esquerda está o sistema real, presente no nível M0 e trata-se de uma instância do modelo do nível M1, que normalmente é código fonte de uma qualquer linguagem de programação.

No modelo que se encontra no nível M1 descreve-se a informação do sistema no nível M0, e a partir do qual o código fonte do sistema é gerado. No nível M2 está o metamodelo e descreve a informação do modelo no nível M1, a partir do qual se define a sintaxe abstrata necessária para escrever os modelos do nível M1. No nível M3 está o meta-metamodelo e descreve os metamodelos que se encontram em M2, bem como o próprio meta-metamodelo em M3. À direita, encontram-se as respetivas linguagens utilizadas para escrever os modelos.

2.2.2 Transformação de Modelos

As transformações de modelos são modelos da tradução de um ou mais modelos de origem em um ou mais modelos alvo [46, 30]. Assim neste processo existe uma relação de mapeamento entre os elementos do modelo fonte e os elementos do modelo alvo (um-para-muitos ou muitos-para-um).

[8] http://www.omg.org/

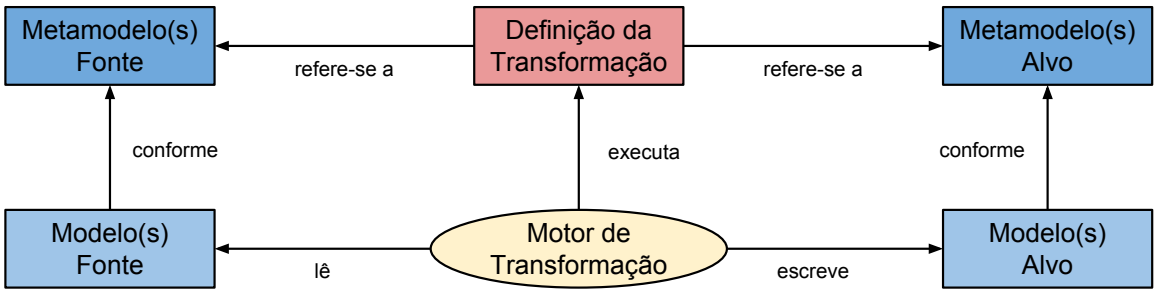


Figura 2.8: Processo de transformação (adaptado de [30])

A figura 2.8 fornece uma visão global do processo de transformação. A transformação é definida de acordo com os metamodelos e executada nos modelos por um motor de transformações.

De uma forma genérica, uma transformação efetua operações simples num modelo alvo, tais como, adicionar ou remover elementos, atualizar propriedades dos elementos e finalmente aceder aos elementos ou às suas propriedades.

Modelo-para-modelo ou modelo-para-texto são as duas principais categorias de transformações. A primeira categoria trata-se de uma transformação entre modelos, ou seja, o resultado é um ou mais modelos que se encontram em conformidade com o seu metamodelo. Na segunda categoria o produto da transformação é texto, onde neste caso poderá ser código numa linguagem de propósito geral ou outro tipo de documentação [45, 30].

Existem diversos tipos de transformações, dos quais para a presente dissertação se destacam:

- **Refinamento de modelos:** Trata-se de transformar uma especificação de alto nível numa descrição de nível inferior;
- **Abstração de modelos:** É o inverso do refinamento. Trata-se de transformar um modelo concreto num modelo mais abstrato;
- **Síntese:** Nesta transformação um modelo é sintetizado numa linguagem bem definida. É utilizada frequentemente em DSLs para transformar modelos em código de uma linguagem de propósito geral;
- **Engenharia Reversa:** É o inverso da síntese. Trata-se de obter uma especificação de nível elevado (modelo) extraído de uma especificação de nível inferior (código fonte);
- ***Rendering*:** Trata-se do mapeamento da sintaxe abstrata na sintaxe concreta, ou seja, a representação da sintaxe abstrata de uma forma textual, gráfica ou outra;
- ***Parser*:** É o inverso do *Rendering*. Trata-se do mapeamento da sintaxe concreta na correspondente sintaxe abstrata;

- ***Refactoring*:** Alterar a estrutura interna do modelo para melhorar alguma característica, sem alterar o seu comportamento;
- **Composição:** Trata-se de integrar modelos que foram desenvolvidos isoladamente num único modelo composto;
- **Sincronização:** Trata-se de garantir a consistência entre modelos. É a transformação que permite manter um ou mais modelos atualizados, com base em alterações efetuadas noutro modelo;
- ***Higher-Order Transformation* (HOT):** Consiste numa transformação que lida com modelos de transformações, ou seja, o próprio artefacto produzido é uma transformação.

2.2.3 Composição de Modelos

2.2.3.1 *Model Merging*

O *model merging* é um tipo de transformação que integra dois modelos fonte num só modelo alvo. O modelo M_A e M_B são instâncias dos metamodelos MM_A e MM_B, respetivamente, e são transformados no modelo alvo M_C que está de acordo com o seu metamodelo MM_C. A transformação é representada por $MERGE_{M_A,M_B \to M_C}$ [20].

Esta transformação é decomposta nas seguintes fases:

- **Comparação:** Verifica-se a correspondência de elementos equivalentes nos modelos fonte. Esta fase evita que sejam produzidos elementos duplicados no modelo alvo;
- **Verificação de conformidade:** Nesta fase, verifica-se se os elementos identificados na fase de comparação estão em conformidade. O principal objetivo é a identificação de potenciais conflitos que possam não permitir efetuar a fusão dos modelos;
- **Fusão:** Efetua-se a fusão dos modelos, produzindo um terceiro modelo;
- **Reconciliação e reestruturação:** Esta fase permite remover inconsistências que possam ter surgido na aplicação da fase anterior, ou seja, eventuais erros são corrigidos e a transformação termina.

O *model merging* apresenta-se como uma transformação essencial para estender o metamodelo das DSLs, quando é necessário adicionar novas características à linguagem já existente. Esta técnica é também utilizada para atribuir semântica às linguagens de modelação para domínio específico [32].

A extensão de modelos é efetuada criando novos modelos compostos a partir de modelos base. Este mecanismo é útil onde existe uma hierarquia de metamodelos. Como um metamodelo também é um modelo, quando o metamodelo é utilizado como uma extensão de um metamodelo base, é então criado um só metamodelo estendido [39].

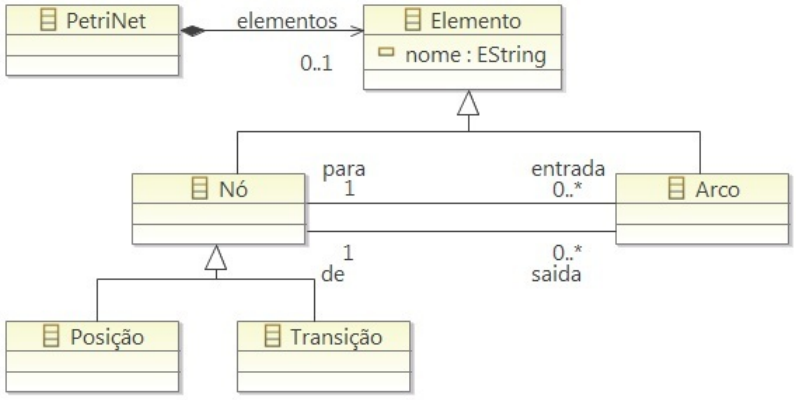


Figura 2.9: Metamodelo base da *Petri net* (adaptado de [39])

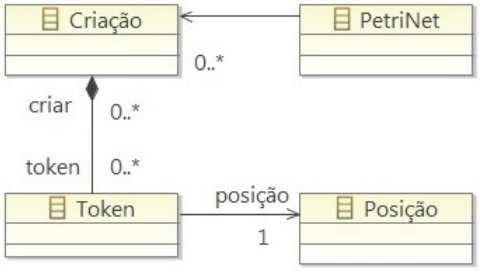


Figura 2.10: Metamodelo extensão para *Petri net* (adaptado de [39])

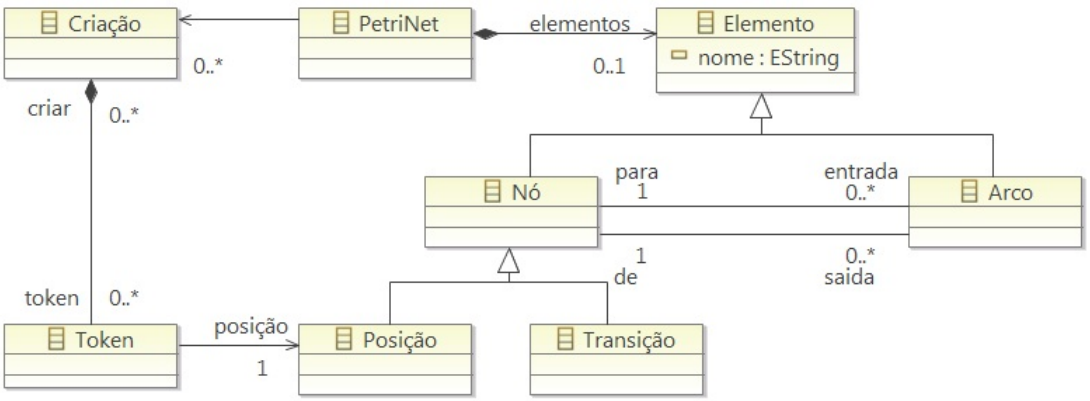


Figura 2.11: Metamodelo composto da *Petri net* (adaptado de [39])

O exemplo apresentado na figura 2.9, 2.10 e 2.11 é baseado no formalismo de *Petri nets*, que se baseia em transições e posições ligados por arcos. O metamodelo na figura 2.9 não permite desenhar *Petri nets* com estados de execução específicos, para tal é necessário estender com o metamodelo da figura 2.10. O resultado da extensão utilizando *model merging* apresenta-se na figura 2.11.

2.2.3.2 *Model Weaving*

O *model weaving* é uma composição que integra dois modelos fonte num só modelo alvo, utilizando ligações. Os modelos podem ter aspetos de modelação diferentes, pois, trata-se de uma operação que estabelece correspondência entre os elementos particulares de cada modelo interveniente.

Para compor utilizando o *model weaving* é necessário criar um tipo especial de modelos, chamado modelos *weaving* que estão conforme o metamodelo *weaving* e definem o tipo de ligações entre modelos que se podem efetuar.

Quando é aplicado às DSLs produzidas através de meta-modelação, os modelos *weaving* contêm um conjunto de ligações entre elementos de um metamodelo e elementos de um segundo metamodelo da linguagem (figura 2.12).

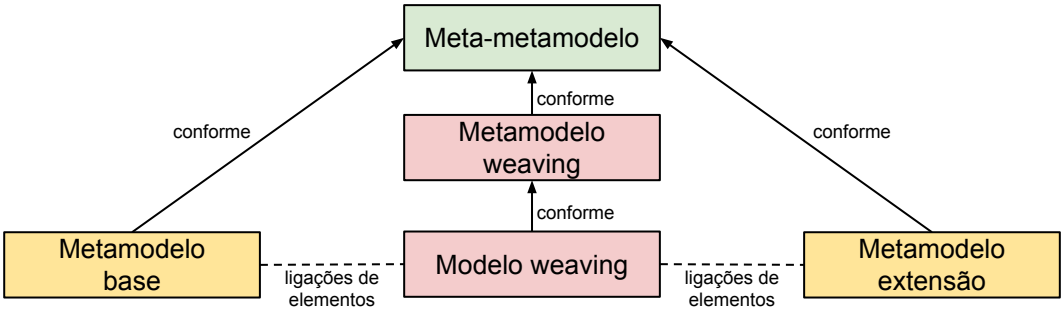


Figura 2.12: *Weaving* de metamodelos

2.3 *Domain Specific Language* (DSL)

Uma DSL é uma linguagem de computador que é otimizada para uma determinada classe de problemas, ou seja, é especializada para representar abstrações de um domínio em particular [37].

Estas linguagens são desenvolvidas tendo em conta as abstrações que os peritos do domínio utilizam e se encontram particularmente habituados, sendo assim, são mais eficientes a representar os problemas do domínio.

Com a utilização de uma DSL evita-se que o programador esteja sujeito a detalhes de implementação que são irrelevantes para o domínio, caso a opção fosse a utilização de uma *General Purpose Language* (GPL), ou seja, o programador não tem de refletir como transformar a informação do domínio de forma a representar corretamente no contexto informático.

As DSLs podem-se apresentar aos seus utilizadores com um aspeto textual ou diagramático. Dependendo do domínio e dos problemas que se pretende codificar deve ser utilizado uma das formas em detrimento de outra.

Independentemente da sua interface, as DSLs permitem aumentar a produtividade,

melhorar a qualidade do produto, promover a reutilização de código e auxiliar especialistas do domínio que não têm conhecimentos avançados em informática.

Quando se pretende desenvolver uma DSL, é necessário atravessar as diversas fases de desenvolvimento demonstradas na figura 2.13 [34, 3]. Em cada fase existem decisões que afetam a solução de forma estrutural. Entre outros fatores, é necessário decidir o paradigma de desenvolvimento que se pretende adotar, com base nas vantagens e desvantagens que cada um apresenta.

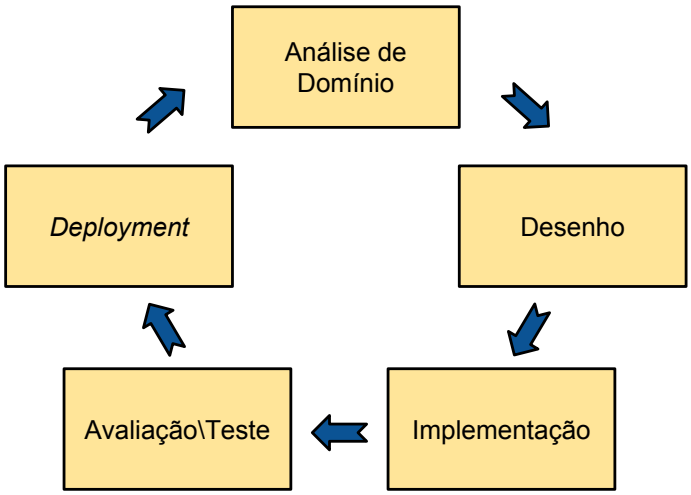


Figura 2.13: Ciclo de desenvolvimento

Na atualidade, é frequente utilizar-se o MDD devido às diversas características apresentadas em 2.2. De seguida serão expostas as principais fases e processos para desenvolver uma DSL com recurso ao desenvolvimento orientado a modelos.

2.3.1 Análise de Domínio

A análise do domínio é uma das fases mais importantes no desenvolvimento de uma DSL, trata-se de perceber com a profundidade adequada, o domínio para o qual a linguagem vai ser desenvolvida.

Esta análise é efetuada com recurso a diversas fontes de informação, nomeadamente, com o estudo da documentação técnica, implementações existentes, requisitos correntes e futuros da linguagem, e através de diálogos com os peritos do domínio. É fulcral um engenheiro de linguagens responsável pelo desenvolvimento da DSL perceber corretamente os conceitos, bem como entender as necessidades e os objetivos que a linguagem tem de incluir. Uma perceção errada nesta fase de desenvolvimento poderá implicar alterações estruturais em fases mais avançadas do mesmo.

É comum efetuar-se uma descrição textual do domínio do problema, uma listagem de conceitos e a sua descrição, um modelo do domínio, um *feature model* [31], uma descrição da plataforma alvo, uma descrição dos utilizadores da DSL e finalmente *user stories*.

O modelo do domínio é essencial para se verificar se o domínio foi entendido na

sua globalidade pelo engenheiro de linguagens. Este permite ter o problema modelado, o que é um ponto de partida para o desenvolvimento do metamodelo da linguagem. Adicionalmente o modelo do domínio também facilita a comunicação entre o engenheiro de linguagens e os especialistas do domínio.

O *feature model* apresenta-se aqui como uma ferramenta essencial para avaliação da variabilidade futura da DSL, ou seja, é possível observar quais os conceitos que a linguagem terá obrigatoriamente de cobrir e aqueles que poderão ser opcionais.

Com o recurso às *user stories* nesta fase de desenvolvimento, proporciona-se ao engenheiro de linguagens a noção do tipo de usabilidade e expressividade que a linguagem terá de fornecer, bem como o tipo de problemas que a DSL terá de permitir codificar.

2.3.2 Desenho da Linguagem

Desenhar uma DSL com recurso à abordagem de desenvolvimento orientado a modelos, é especificar a sintaxe utilizando metamodelos. Um metamodelo, como descrito em 2.2.1 é um modelo do modelo.

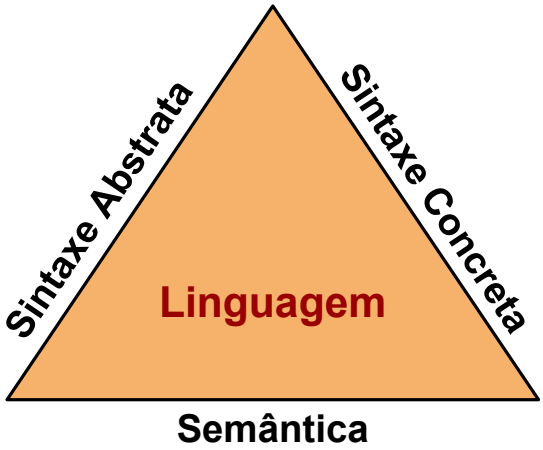


Figura 2.14: Triângulo dourado

Uma linguagem corretamente desenvolvida apresenta uma sintaxe abstrata, uma sintaxe concreta e uma semântica como ilustrado na figura 2.14. A sintaxe abstrata define a construção da linguagem de acordo com as regras, propriedades e relações. Por outras palavras, é uma estrutura de dados que representa os dados semanticamente relevantes e que podem ser expressas por um programa [37]. A sintaxe concreta é a notação que um utilizador interage com a linguagem, podendo ser entendido como o veículo de interação homem máquina [5]. A semântica é o sentido e significado que se atribui a cada construção da linguagem.

Em relação à sintaxe concreta, verifica-se que existem ferramentas que apresentam o metamodelo já desenvolvido, sendo só necessário efetuar as ligações entre a sintaxe abstrata e a sintaxe concreta. Estas ligações são denominadas mapeamento sintático, e são vulgarmente efetuadas com recurso a anotações no metamodelo da sintaxe abstrata

[45].

Devido à separação da sintaxe concreta e abstrata em dois metamodelos distintos é possível encontrar DSLs com ambientes de desenvolvimento completamente diferentes, que têm em comum o metamodelo da sintaxe abstrata [36].

2.3.3 Implementação da Linguagem

2.3.3.1 Geração de Editores Textuais ou Gráficos

Na atualidade, a geração dos editores textuais ou gráficos de uma DSL pode ser efetuada de forma automatizada em diversas ferramentas. Trata-se de uma grande vantagem, pois diminui de forma evidente o tempo de desenvolvimento e torna possível que o engenheiro de linguagens visualize sempre que desejar, o efeito de uma qualquer alteração efetuada nos metamodelos da linguagem.

Quando os geradores automáticos de linguagens não permitem desenvolver a linguagem com a expressividade pretendida, é então necessária uma intervenção de baixo nível, ou seja, os engenheiros informáticos terão de desenvolver o código manualmente.

2.3.3.2 Geração de Código e Outros Artefactos

Frequentemente, quando se desenvolve uma DSL, não se pretende só a representação de conceitos do domínio numa linguagem especializada, pretende-se também obter algum artefacto para posterior utilização. Estes artefactos podem ser modelos (ver na secção 2.2.1) ou texto [37]. Independentemente do artefacto pretendido, este é obtido através de transformações (ver na secção 2.2.2).

Um artefacto normalmente necessário, é código numa linguagem de propósito geral, e este código poderá ser obtido utilizando uma aproximação *visitor-based* ou *template-based* [30]. Na primeira hipótese utiliza-se um mecanismo que percorre a representação interna do modelo e produz um fluxo de texto. A segunda hipótese trata-se de um modelo contendo junções de texto com metacódigo que acede às informações do modelo *source* para realizar a expansão de texto.

2.3.4 Avaliação e Teste da Linguagem

2.3.4.1 Teste da Linguagem

Testar uma DSL, é testar os diversos aspetos da implementação, nomeadamente a sintaxe, as restrições do sistema e a semântica de execução [37].

A sintaxe pode ser testada escrevendo vários programas grandes, complexos e relevantes na linguagem. Caso não seja possível expressar algo, então é porque a linguagem não se encontra completa [37]. Outra forma, é o desenvolvimento de programas errados e verificar se os erros são detetados.

As restrições do sistema podem ser verificadas de forma intencional, ou seja, desenvolve-se código que provoca intencionalmente esses erros e verifica-se se tal é detetado.

A semântica pode ser testada, desenvolvendo programas cujo comportamento se conhece. De seguida, gera-se a sua representação executável e testes unitários que correspondem ao comportamento esperado do programa. Caso o programa não ultrapasse corretamente os testes unitários é porque a semântica da linguagem está errada.

Diversas vezes, testar empiricamente a semântica da linguagem não é suficiente, nesse caso, efetua-se uma verificação formal e assim toda a DSL pode ser testada de uma só vez, recorrendo a vários algoritmos não triviais [37].

Outro tipo de testes podem ser efetuados com o objetivo de avaliar os mais variados fatores, como é exemplo, os serviços fornecidos pelo editor da linguagem. Maioritariamente esses testes são empíricos e baseados em cenários concretos para observar se determinados erros ou problemas conhecidos se verificam.

2.3.4.2 Avaliação da Linguagem

O desenvolvimento de uma DSL apresenta custos que podem ser elevados, no entanto, traz consigo vantagens evidentes que os superam. A qualidade da utilização proporcionada por uma boa usabilidade, maximiza o aumento da produção, pois permite que o utilizador da linguagem seja eficaz, eficiente e se encontre satisfeito com a utilização da interface apresentada pela DSL.

Existem quatro principais formas de avaliar a usabilidade. Efetuar uma avaliação formal utilizando modelos e simulações para tentar prever e medir a usabilidade [4]. Efetuar uma avaliação automática com recurso a um protótipo, para verificar se a interface está de acordo com as linhas orientadoras e os standards [4]. Efetuar uma avaliação empírica onde se testa e valida a interface, propondo aos utilizadores objetivos num determinado contexto de uso, verificando posteriormente se estes os conseguem alcançar [4]. Finalmente utilizando heurísticas que são conduzidas por peritos [4].

A avaliação de uma DSL deve ser efetuada no decorrer das diversas fases de desenvolvimento [4]. Na análise de domínio devem-se identificar corretamente os requisitos de usabilidade para o contexto, tendo em conta o perfil dos utilizadores. Posteriormente no desenho da linguagem devem-se encontrar os atributos importantes para os conceitos do domínio, relacioná-los e encontrar métricas apropriadas. Na implementação e testes devem-se calcular as métricas e verificar se respeitam os requisitos.

Adicionalmente é essencial avaliar as características globais de uma DSL, especialmente quando ainda é possível modificar detalhes incorretos na linguagem, ou seja, antes da implementação se encontrar completamente concluída. Nesse sentido pode-se efetuar uma avaliação empírica de acordo com o processo de avaliação apresentado na figura 2.15 [5].

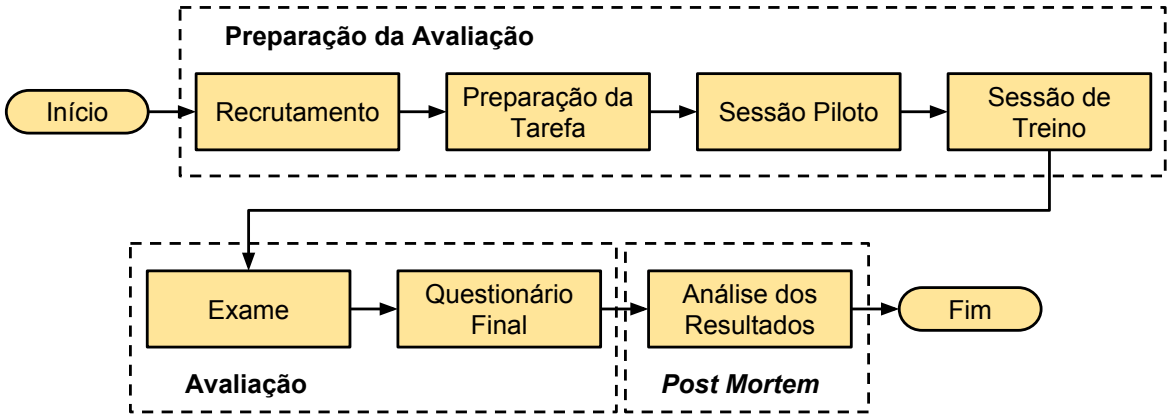


Figura 2.15: Processo de avaliação

Na figura 2.15 a etapa de recrutamento deve ser efetuada com cuidado, pois devem ser escolhidos os utilizadores de acordo com o objetivo da avaliação. De seguida prepara-se a tarefa de forma a surgir o menor número de inconvenientes que possam interferir no processo. Efetua-se uma sessão piloto para descobrir inconvenientes de última hora.

A sessão de treino é o primeiro contacto dos utilizadores com a DSL, onde ficam com uma visão global da mesma, sendo esta, a última etapa de preparação da avaliação. Posteriormente é efetuado o exame onde os utilizadores são expostos a cenários hipotéticos que implementam utilizando a DSL. Nesta etapa são retiradas métricas para posterior análise.

As últimas duas etapas são essenciais para a perceção da qualidade da DSL. No questionário é obtido *feedback*, que permite melhor entender os resultados obtidos no final da etapa de análise dos resultados.

2.3.5 *Deployment* da Linguagem

O *deployment* de uma DSL é idêntico ao *deployment* de qualquer outro *software*. São efetuadas as atividades necessárias que tornam a linguagem disponível para os especialistas do domínio, nomeadamente, preparar a DSL para ser instalada no equipamento informático do cliente.

2.4 *Software Product Line* (SPL)

Uma SPL também pode ser denominada família de produtos de *software* [41] ou família de sistemas [22]. Tratam-se de sistemas de *software* que partilham um conjunto de características que satisfazem as necessidades específicas de um segmento de mercado ou missão.

Um produto numa linha de produtos é desenvolvido a partir de um conjunto de recursos comuns de uma forma determinada, ou seja, não são desenvolvidos de início, nem separadamente [41].

Cada linha de produtos de *software* tem uma plano de produção que especifica como os recursos são utilizados para desenvolver o produto.

Construir um novo produto é maioritariamente um processo de integração de componentes, ao invés de programação exaustiva. Desta forma, é possível aumentar a qualidade do *software* e a produtividade, baseando-se na reutilização de recursos de forma sistemática [28, 41].

Os produtos que pertencem a uma SPL são caracterizados pelos recursos que utilizam, pois podem ser comuns a todos os produtos ou variar entre eles. Esta variabilidade é analisada com recurso a *Feature-Oriented Domain Analysis* (FODA) [31], nomeadamente com a utilização de *feature models* na fase inicial de análise do domínio.

As linhas de produtos de *software* fornecem vantagens económicas a partir da reutilização de componentes, mas necessitam de investimento inicial, na manutenção e evolução dos recursos.

As três atividades essenciais para produção e manutenção de uma linha de produtos são o desenvolvimento dos recursos e produtos, e finalmente a gestão.

O desenvolvimento dos recursos é também chamado *domain engineering* [28, 41]. Os recursos podem ser desenvolvidos e posteriormente utilizados em aplicações. Em alternativa, desenvolvidos para uma aplicação específica e depois extraídos.

O desenvolvimento de um produto utilizando os recursos é chamado *application engineering*, e tem em conta os requisitos pretendidos, a discrição da linha de produtos existente, os recursos necessários para construção e o plano de produção.

Por último, a gestão é responsável pelo fornecimento de recursos financeiros e outros apoios essenciais, coordenação e supervisão de todo o processo. Esta é em última análise, responsável pelo sucesso do esforço despendido na utilização de linhas de produtos.

2.5 Definição de Família de DSLs

Inspirado na definição de SPL, pode-se definir uma família de DSLs como sendo um conjunto de linguagens para um domínio específico, que apresentam um conjunto comum de conceitos chave, mas que adaptam alguns desses conceitos para cumprir a variabilidade dos requisitos.

Trata-se de uma linha de produtos de DSLs, ou seja, é um conjunto de linguagens que têm características comuns. Neste âmbito efetua-se a reutilização de diversos artefactos para o produção de uma nova DSL, nomeadamente no desenvolvimento orientado a modelos pode ser reutilizado de forma completa ou parcial o metamodelo da linguagem.

2.5.1 Abordagens de Desenvolvimento

2.5.1.1 Composição de DSLs

O desenvolvimento de uma família de linguagens pode ser realizado com a composição de várias DSLs, oriundas ou não de aspetos de modelação diferentes.

Um objetivo desta abordagem é a criação de novas linguagens para amortizar o esforço despendido no desenvolvimento das DSLs utilizadas inicialmente na composição.

A composição do novo metamodelo da linguagem é efetuada manualmente pelos engenheiros informáticos como se fosse um puzzle, reaproveitando e adaptando partes de metamodelos. Nesse processo, podem ser auxiliados através da utilização de um *feature model* que descreve os conceitos que são cobertos por cada DSL, as dependências ou incompatibilidades existentes entre DSLs e como o refinamento das mesmas afeta a cobertura de conceitos [26].

A composição reiterada de uma mesma DSL com outras linguagens para domínios específicos é uma prática utilizada quando se pretende adicionar uma funcionalidade chave de uma linguagem hospedeira a uma linguagem convidada [35, 14].

Esta abordagem apresenta diversos inconvenientes, nomeadamente porque uma DSL por definição encontra-se fortemente acoplada ao domínio para o qual foi desenvolvida, logo implica a necessidade de um elevado esforço dos engenheiros informáticos no sentido de efetuar a composição de linguagens oriundas de domínios diferentes [26]. Este esforço deve-se à necessidade de adaptação dos conceitos à realidade da nova linguagem, podendo em última análise ser mais apropriado desenhá-los de raiz.

Outra desvantagem relevante neste processo prende-se com a necessidade de intervenção humana sempre que é necessário gerar uma nova DSL, o que torna o processo demorado e suscetível a erros.

2.5.1.2 Variabilidade Positiva

Em alternativa, é possível a utilização de uma abordagem que recorre à variabilidade positiva do metamodelo da linguagem para criar uma família de linguagens. Genericamente, inicia-se com um metamodelo mínimo comum a todas as linguagens de uma mesma família e posteriormente, adiciona-se de forma composicional as extensões necessárias para obtenção da linguagem pretendida [36].

O processo de extensão pode ser efetuado recorrendo a *model merging* (ver na secção 2.2.3.1) [32, 33] ou *model weaving* (ver na secção 2.2.3.2) [40].

No *model merging*, bem como no *model weaving*, as extensões são ancoradas em pontos previamente especificados através de transformações de composição que realizam essa operação.

Teoricamente é possível estender indefinidamente uma linguagem com recurso à variabilidade positiva, no entanto, é necessário que a composição dos dois metamodelos faça sentido, bem como pertençam ao mesmo domínio [32, 33].

2.5.1.3 Variabilidade Negativa

Contrariamente à abordagem anterior, apresenta-se a variabilidade negativa do metamodelo. Neste caso está definido o metamodelo completo da linguagem. Posteriormente,

removem-se partes do metamodelo de forma a obter a configuração desejada para o novo membro da família de linguagens. [36].

Quando o utilizador pretende criar uma nova DSL, seleciona no *feature model* as funcionalidades que pretende. De seguida, é efetuada a remoção e alteração dos elementos necessários no metamodelo base de forma automatizada com recurso a transformações predefinidas. A última etapa é a geração da linguagem e dos respetivos editores [35].

A variabilidade negativa é apropriada para utilização em situações onde inicialmente se tem conhecimento de todo o domínio a modelar [16]. Esta abordagem possibilita a criação de um número limitado de linguagens da mesma família, de acordo com o número total de combinações que é possível selecionar no *feature model*.

2.5.1.4 Baseado em Compiladores

Finalmente, uma abordagem tradicional baseada em compiladores, ou seja, nesta abordagem um engenheiro informático desenvolve uma dada linguagem e à medida que adiciona ou remove funcionalidades da linguagem cria um novo membro da família de linguagens. Este processo pode ser efetuado utilizando um conjunto de ferramentas específicas [49] ou desenvolvido de forma comum, implementando a linguagem passo a passo.

2.5.2 Estudo e Comparação

Na secção 2.5.1 foram apresentadas as diversas abordagens que foram categorizadas com recurso à literatura científica. Após essa análise, verifica-se que existem quatro abordagens diferentes de desenvolvimento de DSLs.

Abordagem	Orientação do Desenvolvimento	Geração	Extensibilidade
Composição de DSLs	Modelos	Manual	Elevada
Variabilidade Positiva	Modelos	Automática	Elevada
Variabilidade Negativa	Modelos	Automática	Limitada
Baseado em Compiladores	Código	Manual	Elevada

Tabela 2.1: Comparação de abordagens

A tabela 2.1 apresenta a comparação entre as abordagens. Quando se pretende desenvolver uma família de DSLs é necessário ter em consideração o objetivo e o domínio a que se destina, entre outros fatores.

Vulgarmente, pretende-se automatismo na geração da linguagem e uma elevada extensibilidade. Estes dois requisitos apresentam-se disponíveis na variabilidade positiva, que é utilizada em [40], mas só na atualidade a fusão de modelos ficou mais clara através

da contribuição obtida a partir de investigação na área da semântica de linguagens para domínio específico [32, 33].

A variabilidade negativa também apresenta boas características, mas exige inicialmente o metamodelo completo da linguagem, o que em diversos domínios é difícil obter, por ser demasiado extenso ou se encontrar em constante evolução. Este aspeto torna a sua extensibilidade limitada pois são necessárias muitas alterações adicionais para estender o gerador automático de família de linguagens. Nesse caso, como é evidente, seria necessária a intervenção manual de um engenheiro informático.

As restantes duas abordagens, não se apresentam como verdadeiras alternativas para desenvolver uma família de DSLs. A composição de DSLs é uma abordagem que exige um elevado esforço de implementação e é suscetível a erros de desenvolvimento. Por último, a abordagem baseada em compiladores não permite obter partido das caraterísticas do desenvolvimento baseado em modelos, o que se torna uma hipótese a descartar.

Em suma, a variabilidade negativa e positiva são as abordagens mais indicadas dependendo do objetivo que se pretende.

3

Tecnologias Relacionadas

3.1 *Languages Metamodeling Workbench* (LMW)

Trata-se de um ambiente de desenvolvimento composto por um conjunto de ferramentas integradas que facilitam a criação de DSLs. Este ambiente integrado providencia as funcionalidades necessárias para definir o metamodelo, estabelecer relações, propriedades e regras nos modelos.

3.1.1 Linguagens Existentes

3.1.1.1 AToMPM

O AToMPM[1] é um conjunto de ferramentas que se encontra em investigação. Apresenta uma interface web, mas o servidor executa em ambiente Linux. Existe alguma documentação auxiliar e permite que vários utilizadores trabalhem em simultâneo no mesmo projeto. Os modelos e metamodelos são manipulados como grafos. Os metamodelos são representados utilizando o formalismo entidade-relação.

Este conjunto de ferramentas não permite evoluir a linguagem desenvolvida. Do ponto de vista gráfico, o utilizador pode criar ou adicionar facilmente elementos.

Também não existem mecanismos que permitam efetuar composições de modelos e as transformações estão limitadas ao uso de gramática de grafos.

3.1.1.2 *Eclipse Modeling Framework* (EMF) *Graphical Modeling Framework* (GMF)

O EMF[2] é um conjunto de ferramentas de modelação e de geração de código. Os metamodelos são representados com recurso a um formalismo próprio chamado Ecore. O

[1] http://syriani.cs.ua.edu/atompm/atompm.htm
[2] http://www.eclipse.org/modeling/emf/

GMF[3] é um gerador e uma infraestrutura de execução para desenvolvimento de editores gráficos com base no EMF. Tanto o EMF como o GMF, funcionam sobre o Eclipse, sendo assim, podem ser utilizados em Windows, Linux e Mac OS. Existe diversa documentação, e é fortemente apoiado por comunidades abertas de desenvolvimento.

Quando se pretende evoluir a linguagem, basta adicionar elementos, no entanto, caso se remova ou se altere os já existentes, o paradigma da linguagem pode ficar alterado. Neste IDE diversos elementos gráficos podem ser representados, bem como definir grupos de objetos e o seu comportamento, como por exemplo os *containers*. Também é possível alterar os elementos gráficos, bem como adicionar barras de ferramentas.

De forma nativa o EMF/GMF não fornece nenhum mecanismo que permita a composição. Noutro aspeto, é possível instalar *plug-ins* que adicionam linguagens de transformações, por exemplo baseadas em gramáticas de grafos ou *Query/View/Transformation* (QVT).

3.1.1.3 Epsilon

O Epsilon[4] é um conjunto de *software* que inclui o EMF/GMF, EuGENia e diversas linguagens. Algumas das quais, se encontram seguidamente listadas:

- ***Epsilon Object Language (EOL)*:** Trata-se de uma linguagem de programação imperativa para criar, consultar e modificar modelos EMF. Combina o estilo de procedimentos do *JavaScript* (JS) com as capacidades de consulta de modelos da *Object Constraint Language* (OCL);
- ***Epsilon Transformation Language (ETL)*:** É uma linguagem de transformação baseada em regras que permite fazer transformações modelo-para-modelo. Esta linguagem foi construida por cima do EOL, permitindo utilizar as potencialidades dessa linguagem;
- ***Epsilon Validation Language (EVL)*:** Trata-se de uma linguagem de validação de modelos e verificação de consistências. Fornece integração com os editores EMF e GMF;
- ***Epsilon Generation Language (EGL)*:** É uma linguagem de transformação que permite fazer transformações modelo-para-texto baseado numa aproximação *template-based* para gerar código ou outros artefactos textuais;
- ***Epsilon Comparison Language (ECL)*:** Trata-se de uma linguagem baseada em regras que permite efetuar a comparação de diversos tipos de modelos;
- ***Epsilon Merging Language (EML)*:** É uma linguagem baseada em regras para fundir modelos homogéneos ou heterogéneos.

[3] http://www.eclipse.org/modeling/gmp/
[4] http://www.eclipse.org/epsilon/

As principais ferramentas presentes no Epsilon, que são relevantes para a presente dissertação, encontram-se seguidamente listadas:

- **EuGENia:** Trata-se de uma ferramenta responsável por automatizar o processo de geração das DSLs, mantendo o grafismo do GMF e utilizando o formalismo Ecore do EMF;
- ***Workflow*:** É um conjunto de tarefas, que permitem aos implementadores desenvolver complexos *scripts* ANT.

3.1.1.4 *Generic Modeling Environment* (GME)

O GME[5] permite desenvolver uma DSL descrevendo o seu metamodelo. Trata-se de um conjunto de ferramentas que só podem ser executadas em ambiente Windows. Existem manuais disponíveis e tutorais que facilitam a aprendizagem.

De forma idêntica ao EMF, para se evoluir a linguagem, basta adicionar elementos, no entanto, caso se remova ou se altere os já existentes, o paradigma da linguagem pode ficar alterado. O IDE suporta alterar a representação gráfica dos elementos ou conexões.

Para efetuar composições são proporcionadas três opções, o operador de fusão que identifica os pontos de junção através de elementos do metamodelo com o mesmo nome, a herança da interface ou da implementação. As transformações podem ser efetuadas usando a ferramenta *Graph Rewriting And Transformation* (GReAT) ou utilizando o código C++ gerado a partir do metamodelo.

3.1.1.5 MetaEdit+

O MetaEdit+[6] é uma plataforma multi-utilizador que utiliza o formalismo *Graph Object Property Port Role Relationship* (GOPPRR) para definir os metamodelos. O MetaEdit+ pode ser utilizado em Windows, Linux e Mac OS. Existem manuais e esclarecimento de *Frequently Asked Questions* (FAQs) disponíveis.

É possível efetuar a evolução de uma linguagem separando o código escrito manualmente e gerado automaticamente. Nesta plataforma é possível alterar os símbolos ou escolher outros de uma biblioteca. É possível a criação de menus e barras de ferramentas.

Para efetuar composições esta ferramenta é limitada, permitindo só a utilização da biblioteca de metamodelos de linguagens. É possível efetuar transformações alterando o código dos geradores.

[5] http://www.isis.vanderbilt.edu/Projects/gme/
[6] http://www.metacase.com/products.html

3.1.2 Estudo e Comparação

Critério de Avaliação	AToMPM	EMF/GMF	Epsilon	GME	MetaEdit+
Plataformas	●●●	●●●	●●●	●	●●●
Documentação	◖	●●●	●●●	●◖	●◖
Multi-utilizador em concorrência	●	○	○	○	●
Evolução da Linguagem	○	●●◖	●●◖	●●◖	●●◖
Personalização do Grafismo	●●◖	●●●	●●●	●◖	●●◖
Transformação	●◖	●●◖	●●●	●●◖	●◖
Composição	○	○	●	●◖	◖

Tabela 3.1: Comparação de *Languages Metamodeling Workbench*

(Legenda: ○ - Inexistência; ◖ - Cotação parcial; ● - Cotação completa.)

A tabela 3.1 apresenta o sumário da comparação entre as ferramentas estudadas. De forma comum em todas as ferramentas, é possível o desenvolvimento da linguagem com recurso à construção do seu metamodelo.

Como referido anteriormente, o AToMPM ainda se encontra em investigação, tratando-se de uma ferramenta que utiliza um *browser* o que se torna ideal para o desenvolvimento da linguagem num computador com requisitos mínimos. No entanto é possível a visualização das suas fracas prestações ao nível das transformações e suporte para composições. Também se verifica que a documentação é relativamente reduzida o que dificulta a sua utilização.

O GME e o MetaEdit+ encontram-se de uma forma geral equilibrados na cotação. O MetaEdit+ foi desenvolvido para ser comercializado, sendo assim, pode ser utilizado nos três principais ambientes de operação, o que poderá ser uma vantagem considerável quando se pretendem atingir vários tipos de utilizadores alvo. Também permite que diversos utilizadores trabalhem em concorrência no mesmo projeto.

Um aspeto menos favorável, é o facto de não se encontrar tão desenvolvido ao nível das transformações como o GME. Além disso, o GME apresenta melhor suporte para efetuar composição de modelos. É importante referir que só o Epsilon, GME e o MetaEdit+ fornecem suporte nativo para compor modelos, mas, é possível em qualquer uma das ferramentas apresentadas, efetuar composição de modelos com recurso ao desenvolvimento de uma transformação comum.

As ferramentas EMF/GMF e Epsilon apresentam cotações idênticas na tabela de comparação, pois o Epsilon só apresenta diferenças relevantes no conjunto adicional de linguagens e sub ferramentas que o compõem, proporcionando desta forma uma melhor facilidade na sua utilização, bem como a automatização do processo de geração da DSL. Estas duas LMWs apresentam-se em destaque nos resultados observados devido a três principais fatores, sendo eles, a possibilidade da sua utilização em diversos ambientes (Windows, Linux e Mac OS), a abundância de documentação disponível com suporte de uma vasta comunidade de implementadores e entusiastas, e finalmente a possibilidade

de integrar diversos *plug-ins* que fornecem novas potencialidades. Para realizar a composição de modelos, o Epsilon fornece a linguagem EML que permite realizar a fusão de modelos homogéneos ou heterogéneos, apresentando uma vantagem adicional em relação ao EMF/GMF.

De acordo com o estudo apresentado, verifica-se claramente que o Epsilon é a ferramenta apropriada para a realização da prova de conceito, conduzindo posteriormente a um produto pré-comercial. Esta conclusão técnica, apresenta-se em harmonia com as imposições do domínio.

Na aeronáutica o Eclipse[7] é uma plataforma frequentemente utilizada, especialmente devido à sua capacidade de integração com diversas ferramentas, a facilidade de evolução e o elevado suporte da comunidade. De forma complementar, a empresa onde se realizou a presente dissertação, pretendeu que o protótipo fosse desenvolvido utilizando o Eclipse, de forma a integrar o IMADE (ver na secção 8.2).

3.2 Eclipse *Rich Client Platform* (RCP)

O Eclipse RCP é um subconjunto da plataforma Eclipse que permite desenvolver aplicações cliente codificadas em Java, usando *plug-ins* individuais que representam componentes. Adicionalmente fornece o *look and feel* da plataforma original.

É vulgarmente utilizado na indústria como forma de exportação de aplicações sem a necessidade de as implementar de raiz. Trata-se de uma plataforma bastante modular, que permite a integração de *plug-ins* de forma simples.

Uma nova funcionalidade pode ser construída utilizando diversos outros *plug-ins*, sendo assim é possível desenvolver uma aplicação de forma incremental, construindo dependências e interações entre os componentes.

No âmbito da presente dissertação utilizou-se o Eclipse RCP para exportar o protótipo que foi desenvolvido, tornando-o independente da plataforma de desenvolvimento.

[7] http://www.eclipse.org/

4

Análise do Domínio

A análise de domínio foi efetuada com recurso ao estudo de documentação geral, técnica e standards. Foram analisados requisitos de alto nível para a família de DSLs pretendida, bem como um estudo sobre a plataforma alvo. Todo este processo teve o auxílio de peritos do domínio.

Idealmente seria apresentada uma secção onde é exposto o problema do domínio que se pretende solucionar, mas tal já se encontra explicado em 1.3 e de forma aprofundada em 2.1. Em suma, o que se pretendeu foi desenvolver ferramentas que permitissem produzir a estrutura do código de novas aplicações para *aviónica* no contexto IMA. Foi necessário ter em conta a elevada complexidade e os requisitos do domínio.

A aeronáutica apresenta diversos conceitos específicos que foram explicados no decorrer desta dissertação, no entanto na secção 4.1 apresenta-se uma listagem acompanhada de uma breve descrição. Nos anexos, em A.2 apresenta-se o *feature model* do domínio de forma a avaliar a variabilidade das DSLs. Neste caso, bem como no modelo do domínio apresentado nos anexos em A.1 a referência em estudo é o ARINC 653, sendo que a importância está nos mecanismos utilizados nas partições IMA.

4.1 Conceitos do Domínio

De seguida, são descritos diversos conceitos do domínio, em particular, estes encontram-se presentes no *feature model* ou no modelo do domínio.

- ***ARINC 653 Module*:** Módulo de computação *aviónico* IMA que cumpre o standard ARINC 653;

- ***Partition*:** É um programa que é carregado para um espaço de memória num determinado módulo. O sistema operativo instalado num módulo tem o controlo sobre as partições aí instaladas. Cada partição encontra-se isolada de outras;

- ***Process*:** É uma unidade de código na partição que executa concorrentemente com outros processos na mesma partição, ou seja, é uma forma de dividir a partição em uma ou mais tarefas;
- ***Semaphore*:** É uma estrutura usada para providenciar acesso controlado aos recursos da partição, principalmente quando existem dois ou mais processos concorrentes que acedem ao mesmo recurso;
- ***Event*:** Trata-se de um mecanismo que permite a notificação que aconteceu uma determinada condição. Pode existir um dado processo que se encontre à espera da notificação de forma a prosseguir a sua execução;
- ***Blackboard*:** É um meio de comunicação que pode ser utilizado pelos processos da mesma partição para receber e enviar mensagens. Não existem filas de espera, ou seja, cada mensagem nova substitui a anterior;
- ***Buffer*:** É um meio de comunicação que pode ser utilizado pelos processos da mesma partição para receber e enviar mensagens. As mensagens são colocadas numa fila com ordem *First In, First Out* (FIFO);
- ***Port*:** É um recurso que permite que a partição envie e receba mensagens, usando um canal específico;
- ***Queuing Port*:** É um porto que se encontra em funcionamento no modo *Queuing*. Cada mensagem contém dados diferentes, logo, as mensagens são adicionadas numa fila;
- ***Sampling Port*:** É um porto que se encontra em funcionamento no modo *Sampling*. Cada mensagem contém dados idênticos, logo, rescreve a última mensagem existente;
- ***Health Monitor* (HM):** Trata-se de uma função do sistema operativo que monitoriza e reporta falhas no *hardware*, sistema operativo e aplicações;
- ***State*:** Estado de erro que o sistema, módulo ou partição se podem encontrar e a respetiva ação que será executada;
- ***ERROR ID Level*:** Descrição do erro do sistema, módulo ou partição;
- ***ERROR ID Action*:** Descrição da ação que pode ser executada em resposta ao erro ocorrido;
- ***Partition Memory*:** Define as regiões de memória que cada partição tem acesso;
- ***Connection*:** Descreve os canais de comunicação entre partições;
- ***Module Schedule*:** Conjuntos de escalonamentos da execução das diversas partições contidas no módulo;

- ***Partition Schedule*:** Escalonamento da execução de uma partição com recurso a uma ou mais janelas de execução;
- ***Window Schedule*:** Período de tempo enquanto os recursos estão alocados a um dado processo.

4.2 *Feature model*

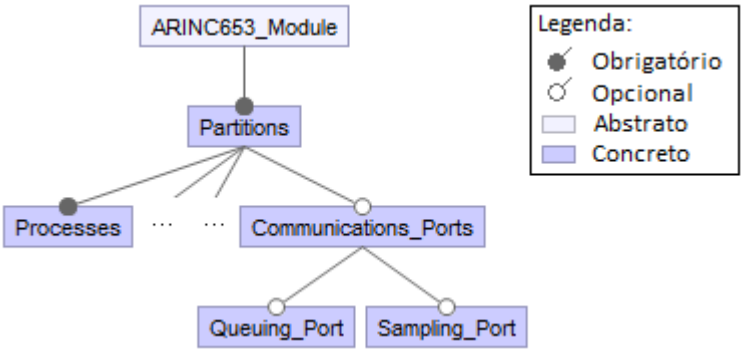


Figura 4.1: Fração do *feature model*

Na figura 4.1 apresenta-se a porção do *feature model* do domínio mais relevante para a dissertação, encontrando-se o *feature model* completo nos anexos em A.2.

O *feature model* é essencial para avaliar a variabilidade das DSLs que serão produzidas, ou seja, através deste é possível observar quais os conceitos que as linguagens têm obrigatoriamente de cobrir e quais os que são opcionais.

4.3 Modelo do Domínio

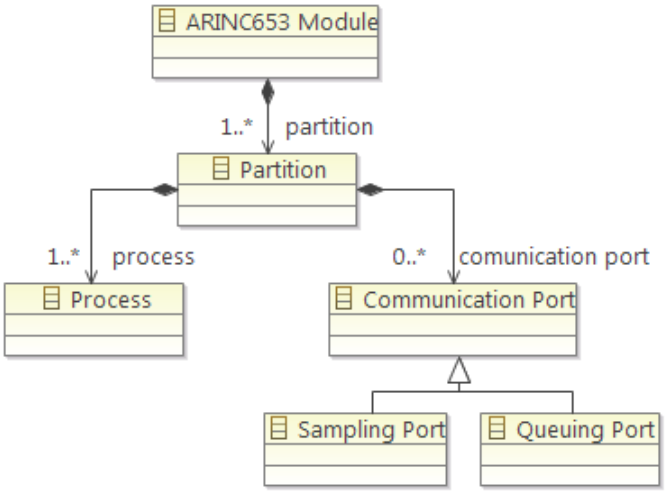


Figura 4.2: Fração do modelo do domínio

Na figura 4.2 encontra-se ilustrada a zona do modelo do domínio que se apresenta mais importante para a dissertação, encontrando-se o modelo do domínio completo nos anexos em A.1.

O modelo do domínio permite ter o domínio modelado com recurso às entidades que o compõem e as relações entre as mesmas.

4.4 *User Story*

O utilizador pretende programar o *Flight Path Viewer* (FPV). Trata-se de uma aplicação *aviónica* responsável pela validação do caminho de voo inserido pelo piloto, quando é efetuada a preparação do voo. A validação é realizada em terra, no entanto, existem estudos oficiais (IMA Segunda Geração) com o objetivo de disponibilizar esta função a bordo das aeronaves. Desta forma, será possível ao piloto alterar a rota por sua iniciativa, caso seja imperativo fazê-lo, mas será sempre necessário validar conflitos entre a nova rota e as diversas rotas de cada avião presente no espaço aéreo.

A aplicação irá obter informação das rotas de outros aviões a partir da base de dados de navegação, com base numa janela temporal indicada pelo piloto. De seguida, cada rota encontrada será confrontada com a nova rota inserida. Caso seja detetado um conflito ou um possível conflito, será indicado no *display* a existência de um problema, bem como toda a informação disponível sobre esse mesmo conflito.

Esta aplicação terá de comunicar com o *Flight Management System* (FMS), de forma a receber informação da nova rota inserida pelo piloto, bem como mostrar a informação necessária. O algoritmo que deteta conflitos divide as rotas obtidas em conjuntos e executa cada um em processos diferentes.

4.5 Perfil de Utilizador

O perfil dos utilizadores que usam as DSLs geradas, tratam-se de especialistas com conhecimento profundo do domínio, bem como de informática avançada. É um utilizador com conhecimento das características do *hardware*, mas também de programação de *software*.

Este público alvo encontra-se habituado a desenvolver aplicações IMA de raiz, o que pretenderam foi uma forma rápida de desenvolver o esqueleto do código, cumprindo as especificações fornecidas sobre a partição onde a aplicação é executada posteriormente. Neste sentido, os utilizadores não pretendiam especificar todos os detalhes da aplicação, ou seja, são utilizadores que a partir de certo ponto no desenvolvimento preferem escrever código ao invés de especifica-lo através de um mecanismo de abstração com nível elevado.

4.6 Plataforma Alvo

A plataforma alvo em última instância é uma aeronave, que contenha módulos de computação IMA. Estes módulos têm instalado um sistema operativo de tempo real que respeita o standard ARINC 653. No sistema operativo existem partições onde estão instaladas as aplicações *aviónicas*.

A plataforma concreta é o sistema operativo AIR (ver na secção 2.1.4.1) e trata-se de um sistema operativo baseado em Linux. As aplicações para serem executadas, têm de ser escritas em código C e respeitar a API definida no standard ARINC 653 denominada de APEX. Estas aplicações são compiladas e só posteriormente poderão ser executadas.

Na figura 4.3 mostra-se o *hardware* onde o sistema operativo e as aplicações executam. Trata-se da placa *Next Generation MicroProcessor* (NGMP)[1].

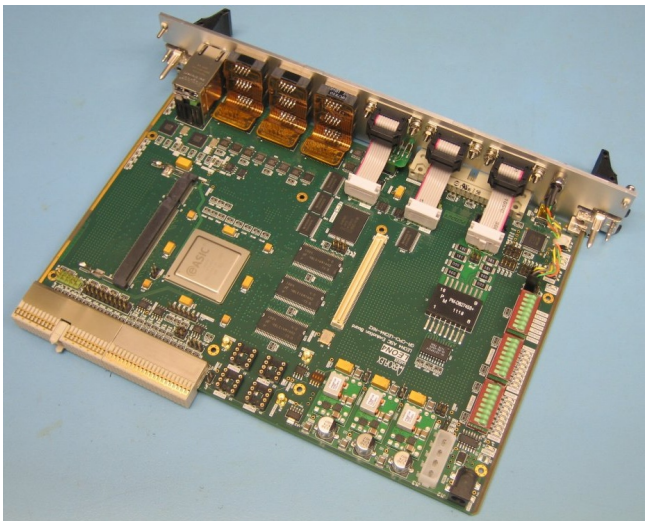


Figura 4.3: Placa NGMP

[1] http://microelectronics.esa.int/ngmp/

5

Solução

5.1 Visão Geral

Com intuito de solucionar o problema apresentado na secção 1.3, foi desenvolvido um protótipo gerador de DSLs. Cada DSL produzida pertence à mesma família e é gerada com recurso a um processo automatizado. Especificamente, este processo demonstra uma abordagem para produzir uma família de linguagens com recurso à variabilidade positiva (ver na secção 2.5.1.2).

As linguagens geradas apresentam conceitos comuns e variáveis (figura 5.1). O primeiro refere-se ao aspeto estrutural e encontra-se claramente relacionado com o domínio. Esta característica faz com que todas as linguagens pertençam à mesma família. O segundo difere entre linguagens e está diretamente associado à configuração do módulo *aviónico* e das partições especificadas nessa mesma configuração.

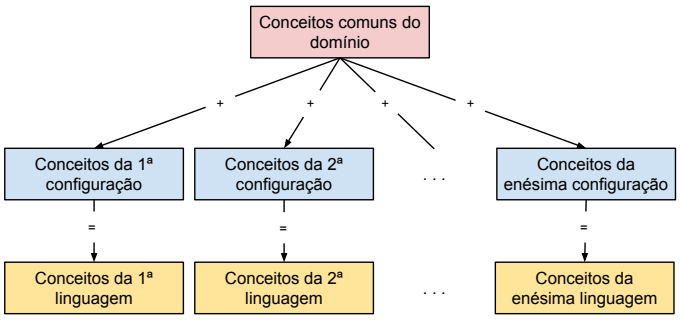


Figura 5.1: Conceitos das Linguagens

Esta solução é proveitosa porque entre os dois aspetos anteriormente referidos, o aspeto estrutural e comum tem uma grande dimensão, sendo assim, não é necessário desenvolver uma DSL completamente distinta para cada configuração de um novo módulo *aviónico* e partição, podendo reaproveitar-se a maioria do esforço despendido no desenvolvimento da estrutura invariável.

Tratando-se de uma dissertação em contexto empresarial foi necessário cumprir vários requisitos propostos pela empresa, tais como:

- **1º Requisito:** As linguagens geradas permitirem somente efetuar operações de acordo com a configuração do modulo *aviónico* e da partição escolhida (limitação na expressividade);
- **2º Requisito:** As linguagens produzidas apresentarem uma interface apropriada para representar os conceitos IMA;
- **3º Requisito:** Apresentarem um nível bom de usabilidade em conformidade com o perfil de utilizador experiente;
- **4º Requisito:** Cada DSL é gerada de forma automática, sem intervenção humana;
- **5º Requisito:** As construções efetuadas pelo programador usando a linguagem serem validadas em *runtime*, confrontando-as com as configurações do módulo;
- **6º Requisito:** Ser possível estender os mecanismos disponíveis para se produzirem novos tipos de artefactos quando necessário.

O primeiro requisito reforça a necessidade de implementar a família de DSLs pois o aspeto variável entre linguagens da mesma família é o fator limitativo da expressividade de cada linguagem.

A interface gráfica apropriada para representar os conceitos IMA (2º requisito) foi aperfeiçoada com recurso às opiniões dos profissionais do domínio durante o desenvolvimento da ferramenta. Nesse sentido os profissionais foram expressando as suas opiniões e consequentemente, sempre que necessário foram efetuadas algumas correções. Este processo levou adicionalmente ao cumprimento do terceiro requisito, ou seja, as linguagens produzidas apresentam a usabilidade indicada para um utilizador experiente conforme é demonstrado com a avaliação de usabilidade efetuado e descrito na secção 7.

A geração automática das linguagens (4º requisito) e a extensibilidade dos mecanismos de geração de artefactos (6º requisito) beneficiaram diretamente do desenvolvimento orientado por modelos utilizado na solução implementada. De forma intencional, foi prioritário evitar a intervenção humana em todo o processo para minorar a necessidade de conhecimento adicional por parte do utilizador quando este pretende manipular a ferramenta. Em suma evitar o número de etapas que é necessário percorrer para gerar uma nova DSL.

Para cumprir o 6º requisito basta desenvolver novas transformações dos modelos de forma a produzir o artefacto desejado. Neste caso será necessário a intervenção de um engenheiro informático, mas será um aspeto pontual e esporádico na fase de manutenção presente no ciclo de vida do *software*. Nesta dissertação produz-se unicamente o *template* responsável por gerar o código C, mas com o desenvolvimento do mesmo, prova-se empiricamente que é possível produzir outro tipo de artefactos.

A validação das construções (5º requisito), é efetuado com recurso ao *Epsilon Validation Language* (EVL) disponível no ambiente de desenvolvimento Epsilon (ver na secção 3). Desta forma é possível verificar a existência de erros enquanto o implementador se encontra a desenvolver a estrutura da aplicação *aviónica* usando a DSL. Assim quando um problema é detetado, o programador é alertado e poderá corrigi-lo.

Do ponto de vista do utilizador experiente que pretende desenvolver uma aplicação *aviónica* para executar numa dada partição, o processo torna-se bastante simplificado. Em primeiro lugar é necessário iniciar a execução do gerador de DSLs, selecionar o ficheiro XML que contém a configuração do módulo, bem como a partição que irá executar essa aplicação. Posteriormente o gerador produz a DSL que poderá de seguida ser utilizada.

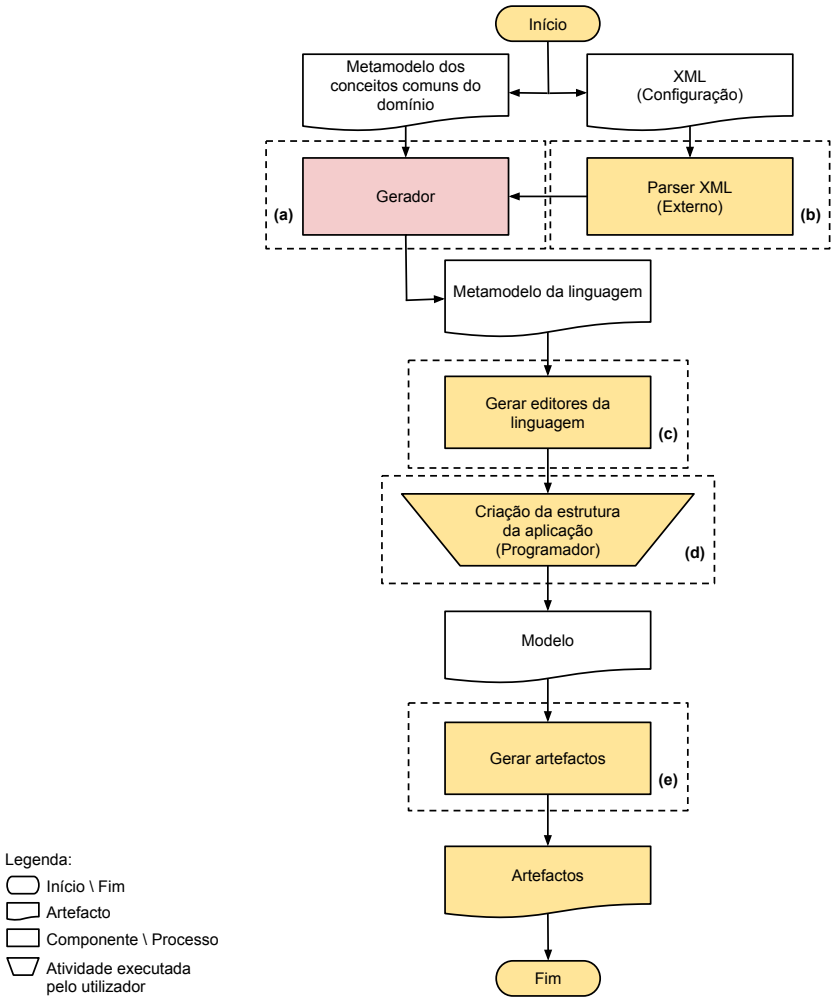


Figura 5.2: Processo genérico da solução

Na figura 5.2 apresenta-se o processo genérico de alto nível da solução implementada considerando também o contexto de utilização. A solução baseia-se no uso de MDD, sendo assim, para produzir uma linguagem é necessário desenvolver o metamodelo da mesma.

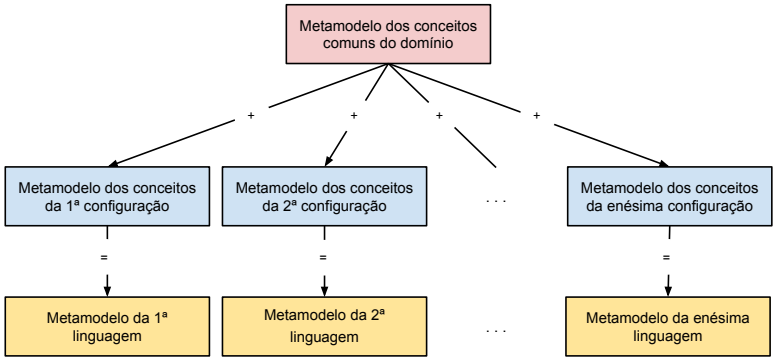


Figura 5.3: Construção dos metamodelos das linguagens

O metamodelo de uma linguagem pertencente à família de DSLs, é construído em três fases (figura 5.3). Na primeira fase foi construído o metamodelo dos conceitos comuns a todas as linguagens que descreve o domínio e as características obrigatórias que as linguagens têm de apresentar (doravante designado por: Metamodelo dos conceitos comuns do domínio). As características foram identificadas com recurso ao *feature model* presente nos anexos em A.2. Este metamodelo foi desenvolvido à priori no momento da produção do protótipo, fazendo parte integral da solução, é imutável, e trata-se de um dos artefactos de entrada do gerador de linguagens. O metamodelo dos conceitos comuns do domínio encontra-se ilustrado nos anexo em A.3.

A segunda fase acontece quando se pretende gerar uma nova DSL, sendo necessário ter o ficheiro de configuração XML. Nesta fase é gerado um novo metamodelo adicional considerando-se a configuração do módulo *aviónico* e da partição escolhida (doravante designado por: Metamodelo dos conceitos de configuração). Este metamodelo funciona como extensão do metamodelo dos conceitos comuns do domínio sendo ambos fundidos em pontos previamente determinados utilizando a técnica apropriada para o efeito (ver na secção 2.2.3.1). A fusão dos metamodelos corresponde à terceira fase. Um exemplo do metamodelo dos conceitos de configuração encontra-se ilustrado na figura 5.24.

A geração do metamodelo da linguagem é realizada no gerador e está presente na figura 5.2(a). Para o efeito, o metamodelo dos conceitos de configuração é produzido com recurso à informação proveniente do *Parser* XML na figura 5.2(b). Posteriormente o gerador faz a fusão do metamodelo dos conceitos comuns do domínio com o metamodelo dos conceitos de configuração, originando assim o metamodelo da linguagem. Para ilustrar um exemplo deste metamodelo considera-se o metamodelo da figura A.3 fundido com o metamodelo da figura 5.24 pela entidade *ARINC653*.

O gerador é responsável por gerar outros artefactos, nomeadamente o ficheiro de configuração do editor gráfico, as regras de validação do editor e o *template* responsável por gerar o código C (artefactos não ilustrados na figura).

A etapa seguinte está explicitada na secção 2.3.3.1. Trata-se da geração dos editores gráficos da linguagem e encontra-se ilustrado na figura 5.2(c). Esta operação é efetuada

com o EuGENia, ferramenta que pertence ao Epsilon (ver na secção 3.1.1.3).

As seguintes etapas a partir de 5.2(d) estão relacionadas com a utilização da DSL gerada. Em 5.2(d) são utilizadas as regras de validação produzidas pelo gerador para validar as construções efetuadas pelo utilizador. Em 5.2(e) é utilizado o *template* para gerar o esqueleto de código C. Este, será manualmente preenchido à posteriori com o código específico da aplicação *aviónica* a ser desenvolvida pelo utilizador (fora do âmbito desta dissertação).

5.2 Solução Técnica

No processo genérico da solução apresentado na figura 5.2, mostra-se que o metamodelo da linguagem é produzido pelo gerador, no entanto outros artefactos são necessários para produzir uma linguagem completamente funcional que apresente uma usabilidade correta, bem como ser capaz de produzir os artefactos exigidos (neste caso, o esqueleto da aplicação *aviónica* em código C).

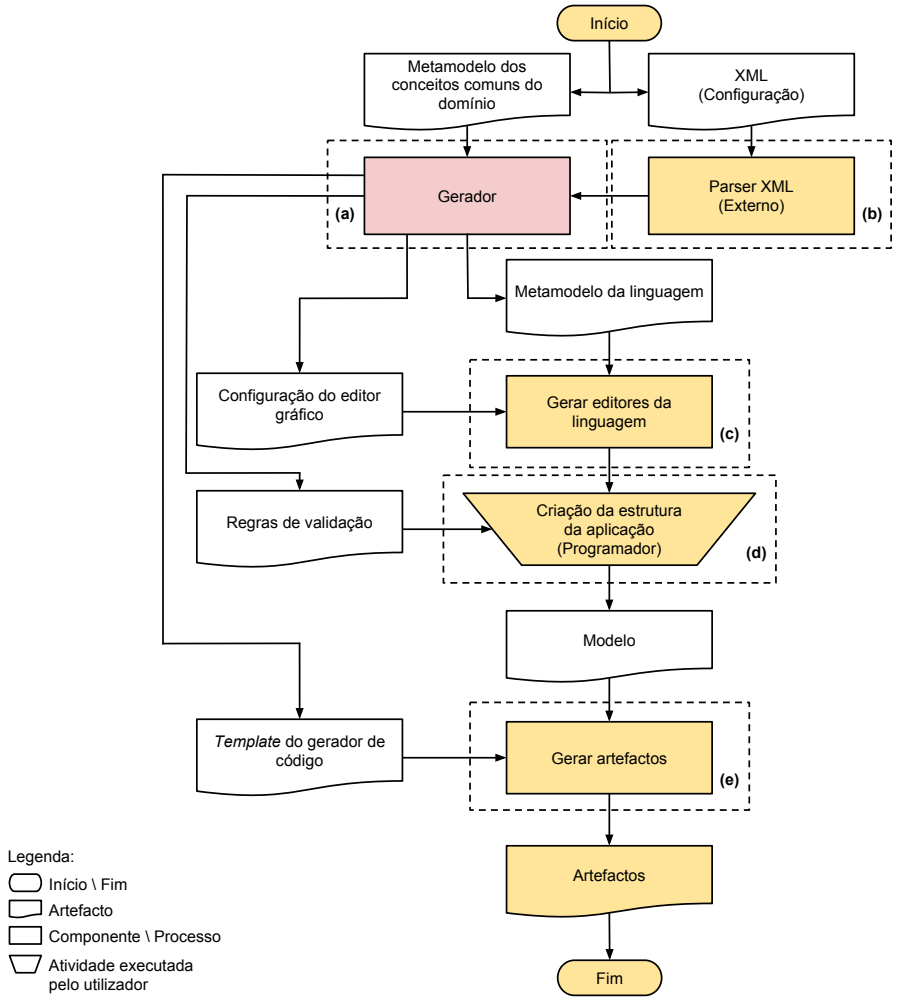


Figura 5.4: Processo da solução

A figura 5.4 demonstra o processo completo da solução. Verifica-se que o gerador é também responsável por gerar a configuração do editor gráfico, as regras de validação e o *template* que produz o código C. A configuração do editor gráfico permite aperfeiçoar o aspeto da ferramenta, melhorando significativamente a usabilidade. Este artefacto é utilizado em conjunto com o metamodelo da linguagem, no momento de produzir os editores gráficos.

As regras de validação permitem alertar o utilizador dos erros que este possa estar a cometer quando usa a DSL, ou seja o artefacto é utilizado no momento de execução da linguagem. O *template* gerador de código C permite produzir o resultado final pretendido, sem este, a própria DSL não faria sentido existir. Este último artefacto é executado quando o utilizador/programador pretende exportar o seu trabalho.

5.2.1 Módulo Gerador de DSLs

O módulo gerador é o elemento central desta abordagem, pois é responsável por gerar todos os artefactos necessários para produzir uma DSL da família de DSLs. A sua composição interna é ilustrada na figura 5.5.

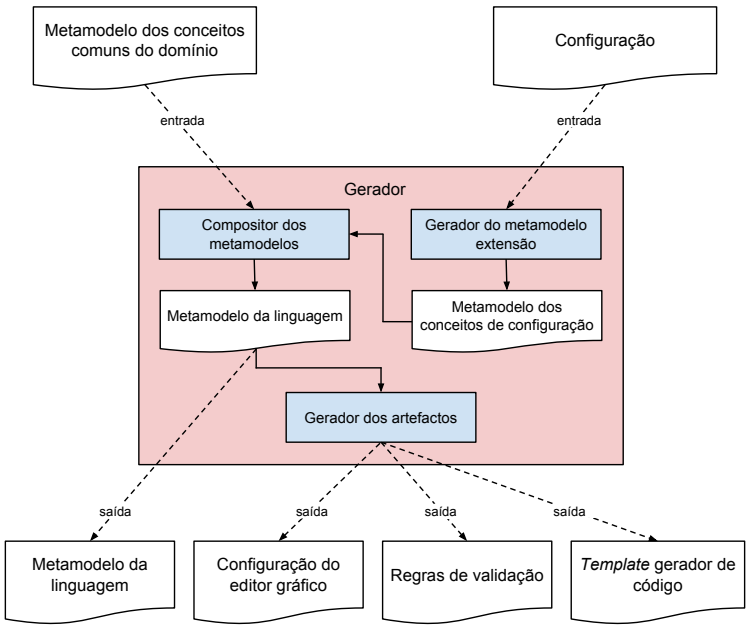


Figura 5.5: Módulo gerador dos artefactos

O gerador recebe de entrada dois tipos de informação, a configuração da partição *aviónica* e o metamodelo dos conceitos comuns do domínio. Este metamodelo é uma fração do metamodelo abstrato da linguagem com anotações (ver na secção 2.3.2).

Este módulo pode ser dividido em duas unidades lógicas, a primeira efetua as operações necessárias para gerar o metamodelo da linguagem e é composta pelos sub módulos *Gerador do metamodelo extensão* e *Compositor dos metamodelos*. A segunda unidade

lógica gera os restantes artefactos necessários com recurso à informação presente no metamodelo da linguagem e trata-se do *Gerador dos artefactos*. Os três sub módulos estão sombreados a azul na figura 5.5.

5.2.2 Geração do Metamodelo da Linguagem

No processo de geração do metamodelo dos conceitos de configuração, que ocorre no sub módulo *Gerador do metamodelo extensão*, efetuam-se operações ao nível do metamodelo da linguagem. Este metamodelo está representado usando o formalismo Ecore, logo é necessário manipular elementos do meta-metamodelo do Ecore, que se encontra detalhadamente descrito em [19]. Alguns elementos são: *EClass*, *EAttribute*, *EReference*, entre outros.

Nesta fase criam-se e organizam-se os novos elementos no metamodelo dos conceitos de configuração, que se encontra inicialmente vazio. A configuração da partição pertencente ao módulo *aviónico* fornece a informação necessária para desenvolver a extensão (principalmente o nome e tipo dos portos de comunicação que é possível utilizar pela partição).

Para concluir o metamodelo dos conceitos de configuração, é necessário efetuar anotações no metamodelo parcial da sintaxe abstrata da linguagem até aqui construído. Estas anotações mapeiam o metamodelo da sintaxe concreta do EMF no metamodelo parcial da sintaxe abstrata (ver na secção 2.3.2). As anotações são efetuadas paralelamente com o desenvolvimento do metamodelo parcial da sintaxe abstrata. Em suma o metamodelo dos conceitos de configuração é uma fração do metamodelo abstrato da linguagem com anotações.

No sub módulo *Compositor dos metamodelos* efetua-se a composição do metamodelo dos conceitos de configuração com o metamodelo dos conceitos comuns do domínio. Desta operação resulta o metamodelo completo da DSL. Para o efeito utiliza-se uma transformação de composição denominada *Model Merging* (ver na secção 2.2.3.1).

5.2.3 Geração dos Artefactos

A configuração do editor gráfico, as regras de validação e o *template* gerador de código C são produzidos no sub módulo *Gerador dos artefactos*. A geração destes três artefactos é realizada com recurso a transformações modelo-para-texto, utilizando-se uma abordagem *template* que contempla as possíveis variações no metamodelo da linguagem. Os *templates* só geram resultados para os elementos existentes no metamodelo.

Alguns editores gráficos disponibilizados pelas várias LMWs existentes, são configuráveis e permitem escolher algumas características, como por exemplo, a localização das barras de ferramentas e dos seus botões, renomear elementos visuais, entre outros.

Visto ter-se optado pelo Eclipse Epsilon, é possível entre outras possibilidades, definir em que grupo de elementos da palete se coloca um determinado elemento gráfico. Elemento este, que posteriormente pode ser utilizado no editor da linguagem gerada.

Para produzir este artefacto, basta conhecer os elementos que existem no metamodelo e que têm obrigatoriamente de aparecer na palete do editor gráfico, nomeadamente as instruções ARINC 653 que fazem parte do metamodelo dos conceitos comuns do domínio. Caso a partição correspondente à DSL produzida não disponha de *Queuing Ports* ou *Sampling Ports*, então as instruções ARINC 653 para lidar com as mesmas não aparecerão no editor gráfico.

As regras de validação e o *template* gerador de código C, são também produzidos com recurso a um *template* específico para cada um. Os artefactos gerados para serem executados, necessitam de informação adicional que provêm do exterior e referem-se à configuração da partição *aviónica* (informação que varia entre as diferentes DSLs da família), logo é necessário garantir com eficiência o acesso a essa informação em *runtime* (ver na secção 6.1.2).

De forma idêntica à configuração do editor gráfico, caso a partição correspondente à DSL produzida não disponha de *Queuing Ports* ou *Sampling Ports*, não serão então, produzidas regras de validação referente aos portos, bem como não será produzido parte do *template* gerador de código C.

O artefacto *template* gerador de código C tem a particularidade de não ser o artefacto final, mas ele próprio é produzido com recurso a um *template*, ou seja, trata-se de uma transformação produzida por uma transformação. Logo este artefacto é uma *Higher-Order Transformation* (HOT).

5.3 Aplicação da Solução (Domínio Simplificado)

A solução para o problema industrial apresentado nesta dissertação foi resolvido com recurso à noção de família de DSLs, onde as próprias linguagens produzidas são maioritariamente idênticas mas também parcialmente distintas entre si. Nesta secção demonstra-se a aplicação desta solução num domínio simplificado. Desta forma contribui-se para a clarificação da solução adotada, bem como, mostra-se que é possível aplicar noutros domínios.

5.3.1 Análise do Domínio

Com a presente análise pretende-se facilitar a compreensão do domínio simplificado que será usado na aplicação da solução, no entanto, visto o foco desta dissertação não ser este domínio, apresenta-se uma análise pouco exaustiva.

5.3.1.1 Domínio do Problema

O domínio é um robô capaz de se mover no mundo real através da utilização das suas rodas, detetar se embateu em algum objeto através do seu para-choque e controlar a intensidade de uma luz.

As rodas do robô movem-se sempre no mesmo sentido, ou seja, só é possível mover o robô para a frente. Nenhuma das rodas tem direção e a única forma de orientar o robô para a esquerda ou direita é mover controladamente só as rodas de um dos lados.

O para-choque encontra-se situado na parte frontal do robô e quando deteta alguma colisão fica parado à espera da intervenção do utilizador sem efetuar nenhuma tarefa. Este sensor é usado como medida de segurança e não pode ser utilizado de outra forma.

O utilizador só pode usar as rodas e a luz que se encontra na parte superior do robô. Esta pode estar desligada ou apresentar a intensidade mínima, média ou máxima.

Para programar o robô é necessário ligá-lo a um computador através de um cabo e efetuar o *upload* de um ficheiro XML com as ações que o robô tem de desempenhar. A aplicação responsável pelo *upload* é externa e independente da DSL.

5.3.1.2 Conceitos do Domínio

Na listagem seguinte, são descritos diversos conceitos relacionados com o domínio em particular, estes são importantes para interpretar o *feature model* na figura 5.6 e o modelo do domínio na figura 5.7.

- ***Corpo*:** Conjunto total do *hardware* que compõe o robô;
- ***Rodas*:** Permitem que o robô se desloque e movem-se só para a frente;
- ***Para-choque*:** Sensor que deteta uma colisão frontal e imobiliza o robô (não é possível interagir, nem ser controlado pelo utilizador);
- ***Luz*:** Iluminação localizada no topo do robô e pode estar desligada ou apresentar três níveis distintos de intensidade luminosa.

5.3.1.3 *Feature model*

O robô é caracterizado pelo *feature model* da figura 5.6.

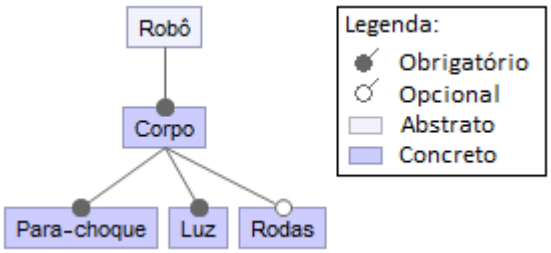


Figura 5.6: *Feature model* do robô

Com uma análise do *feature model* verifica-se que poderão existir várias versões do robô. Cada robô terá obrigatoriamente para-choque e luz, no entanto, poderá existir uma versão com rodas e outra sem rodas dependendo da configuração.

5.3.1.4 Modelo do Domínio

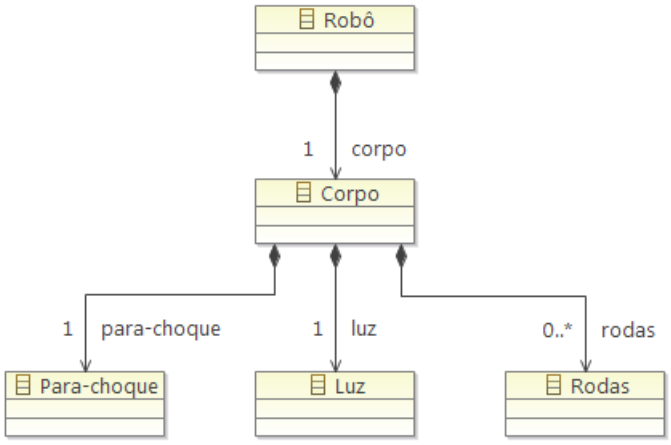


Figura 5.7: Modelo do domínio simplificado

Na figura 5.7 é mostrado o modelo do domínio onde estão representadas as entidades e a cardinalidade das relações entre as mesmas.

5.3.1.5 User Story

O utilizador pretende programar o robô por forma a que este realize, uma única vez, um percurso. Este percurso inclui andar em frente 50 unidades de tempo, mas só ao fim de 10 unidades de tempo é que eleva a luminosidade para o máximo. A luz mantém-se com essa intensidade durante 20 unidades de tempo.

5.3.2 Configurações do Robô

Considerando como base o *feature model* do robô na figura 5.6, e supondo hipoteticamente que o seu fabricante poderia desejar produzir novas variantes da configuração física do mesmo, alterando o número e a posição das rodas, poderia então, surgir as configurações com uma, duas, três, quatro ou mais rodas.

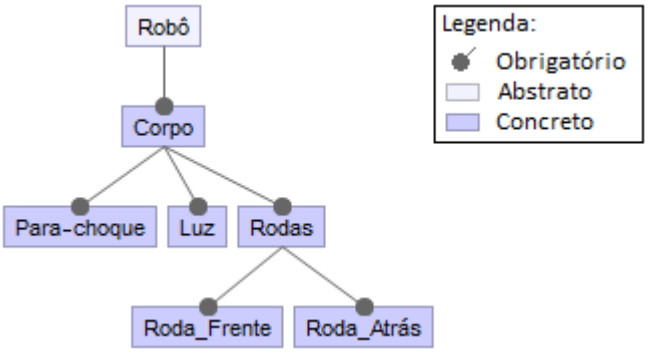


Figura 5.8: *Feature model* do robô com duas rodas

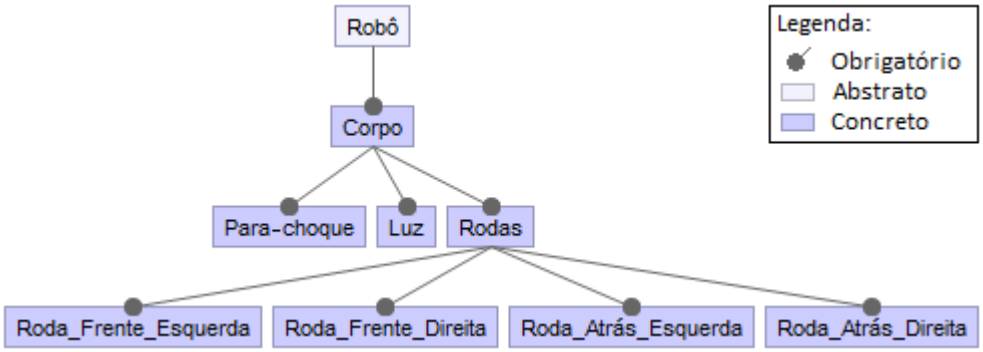


Figura 5.9: *Feature model* do robô com quatro rodas

Supondo duas versões do robô distintas, um robô com duas rodas dispostas em linha caracterizado pelo *feature model* da figura 5.8 e um robô com quatro rodas dispostas duas de cada lado caracterizado pelo *feature model* da figura 5.9. Verifica-se que a cardinalidade da entidade Rodas é relevante se forem consideradas diversas configurações do *hardware* que possam ainda não existir e/ou estar definidas, mas sabe-se que só esse elemento da configuração física do robô poderá mudar. Utilizando uma abordagem de representação da cardinalidade descrita em [29], o novo *feature model* do robô pode ser observado na figura 5.10.

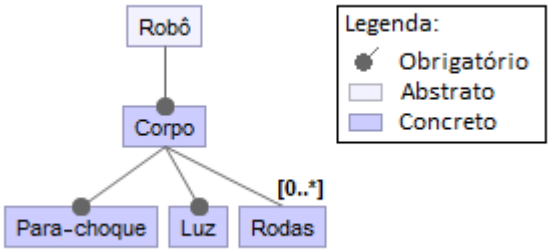


Figura 5.10: *Feature model* com cardinalidade do robô

5.3.3 Família de DSLs para o Domínio Simplificado

Em harmonia com a análise de domínio efetuada, e tendo especial consideração os três *feature model* das figuras 5.8, 5.9 e 5.10, foram desenvolvidas duas DSLs separadamente. Ambas usando as ferramentas fornecidas pelo Eclipse Epsilon (ver na secção 3.1.1.2). De igual forma, os metamodelos das linguagens foram desenvolvidos usando o formalismo Ecore.

Uma DSL é para o robô de duas rodas dispostas em linha e a segunda DSL é para o robô de quatro rodas dispostas duas de cada lado. De seguida vai ser demonstrado como as duas linguagens poderiam ter sido desenvolvidas usando a geração automática de DSLs da mesma família.

Visto tratar-se de duas DSLs com intuito de demonstrar a aplicação da abordagem, só

foram gerados os editores gráficos dessas mesmas linguagens, ou seja, não foi desenvolvido o gerador de DSLs para o robô. Por consequência, também não foram desenvolvidas as configurações do editor gráfico, as regras de validação do editor, nem o *template* responsável por produzir o XML para ser carregado no robô. No entanto, demonstra-se através de pseudocódigo as operações necessárias para gerar esses artefactos, bem como, exemplos de possíveis resultados obtidos.

5.3.3.1 Editor Gráfico das DSLs

A figura 5.11 apresenta o metamodelo da DSL para o robô com duas rodas em linha. O aspeto gráfico do mesmo apresenta-se na figura 5.12, onde foi representada a *user story* descrita na secção 5.3.1.5.

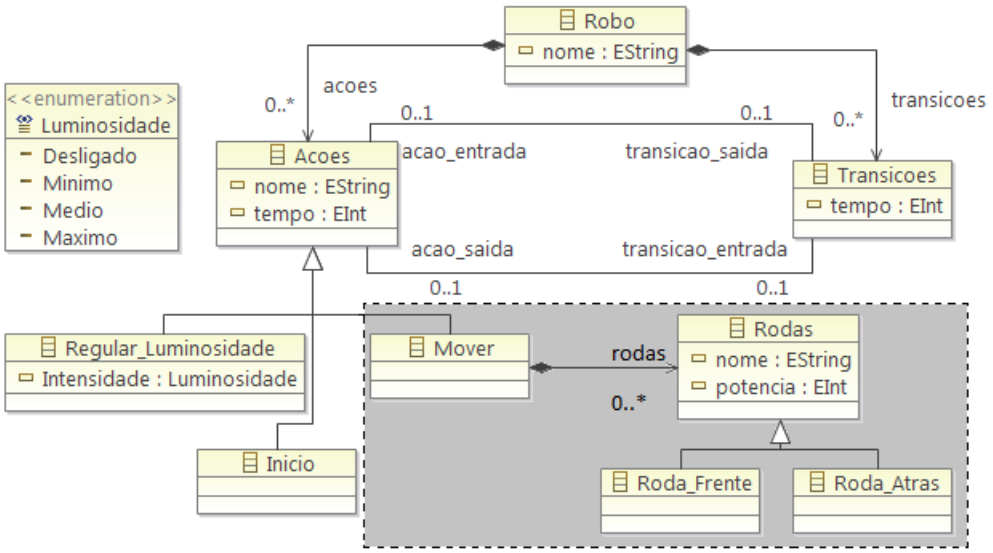


Figura 5.11: Metamodelo da DSL para o robô com duas rodas

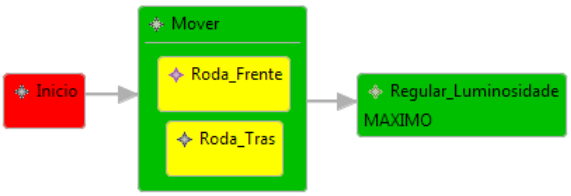


Figura 5.12: DSL para o robô com duas rodas

As entidades *Inicio*, *Regular_Luminosidade*, *Mover*, *Roda_Frente*, *Roda_Atras* e *Transicoes* que estão presentes no metamodelo, aparecem no editor. Trata-se de uma linguagem que adota uma perspetiva estado-transição, ou seja, neste caso, existe uma transição que tem uma ação antecedente e outra ação consequente.

O *Inicio* é a entidade que referencia o ponto inicial a partir do qual o utilizador definiu o comportamento do robô. O fim é intrínseco e verifica-se quando uma ação não tem uma

transição consequente.

O *Mover* indica ao robô para mover as rodas em frente. As rodas que se movem são as que se encontram no interior do retângulo verde e têm a cor amarela. Finalmente *Regular_Luminosidade* permite alterar a intensidade da luz.

As transições e ações, têm uma propriedade *tempo*, que permite no caso das transições definir um atraso em relação à execução da próxima ação face ao início da execução da ação anterior. No caso das ações define o tempo que esta irá executar.

A figura 5.13 apresenta o metamodelo da DSL para o robô com quatro rodas dispostas duas de cada lado. O aspeto gráfico da mesma apresenta-se na figura 5.14, onde também foi representada a *user story* descrita na secção 5.3.1.5.

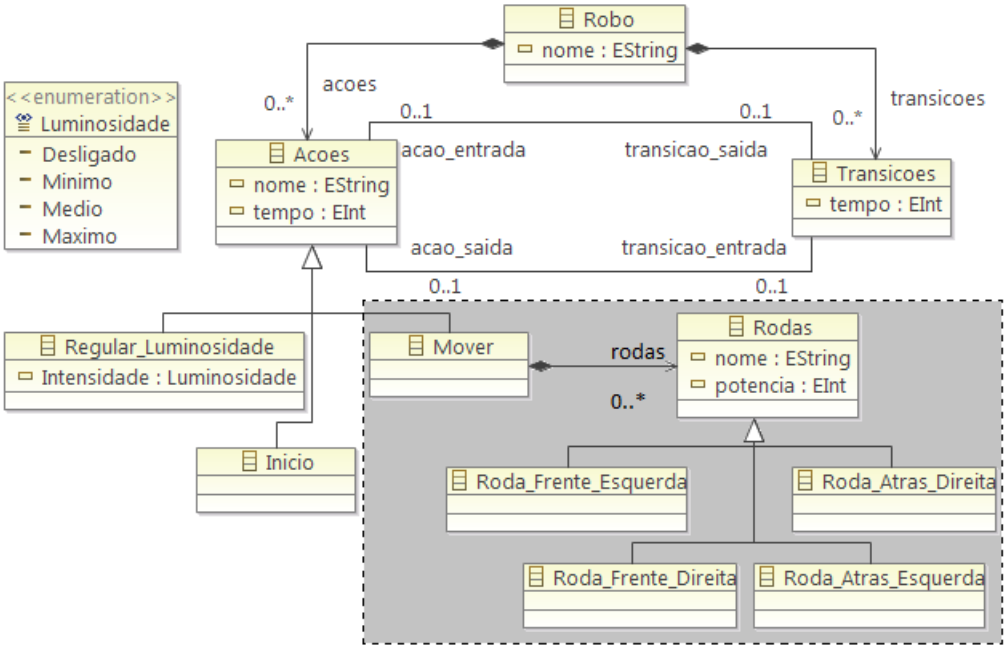


Figura 5.13: Metamodelo da DSL para o robô com quatro rodas

Figura 5.14: DSL para o robô com quatro rodas

A linguagem para o robô com quatro rodas apresenta a mesma usabilidade que a linguagem para o robô com duas rodas, mas as entidades *Roda_Frente* e *Roda_Atras* são substituídas pelas entidades *Roda_Frente_Esquerda*, *Roda_Frente_Direita*, *Roda_Atras_Esquerda* e *Roda_Atras_Direita*.

É expectável que o robô com quatro rodas possa executar comportamentos adicionais face ao robô de duas rodas, por exemplo virar para a esquerda (mover só as rodas do lado direito) ou virar para a direita (mover só as rodas do lado esquerdo). Estes novos comportamentos não afetam o modelo de interação da DSL, pois neste caso, a forma de usar as rodas para especificar um determinado comportamento é definida pelo utilizador.

Nos metamodelos das DSLs apresentados nas figuras 5.11 e 5.13, observa-se uma zona propositadamente sombreada. Trata-se da fração do metamodelo que é ligeiramente diferente entre as duas linguagens, no limite, caso o robô não tenha rodas, esta fração será vazia. Assim, fazendo uma ponte com a solução apresentada na secção 5.2, pode-se considerar de forma genérica que a zona sombreada é a zona correspondente ao metamodelo dos conceitos de configuração. De forma semelhante, a zona não sombreada é a zona correspondente ao metamodelo dos conceitos comuns do domínio.

5.3.3.2 Geração dos Artefactos

Nesta secção exemplifica-se com recurso a pseudocódigos como seriam as transformações responsáveis por gerar a totalidade da configuração do editor gráfico, das regras de validação e do *template* gerador de XML.

Para desenvolver estes artefactos é essencial saber quais são as entidades especificas da configuração. Essas entidades já foram obtidas anteriormente para gerar o metamodelo dos conceitos de configuração (supõem-se que foi implementado a solução de família de DSLs para o robô, ou seja existe um gerador), logo são conhecidas.

Adicionalmente pode ser necessária alguma informação exterior referente à configuração do robô, que pode ser obtida através de diversos métodos distintos e não será aprofundada. Todas as transformações são modelo-para-texto e a informação de saída é escrita em ficheiros separados.

Alguns editores permitem personalizar o aspeto gráfico, para tal é necessário discriminar essas configurações em algum formalismo normalmente é utilizado código em alguma linguagem de programação, neste caso o exemplo usa a linguagem *Epsilon Object Language* (EOL).

A transformação responsável por produzir a configuração do editor teria o comportamento descrito no pseudocódigo da figura 5.15. Pretende-se colocar os elementos gráficos no grupo correto da palete, ou seja, a barra lateral direita (palete) teria um novo grupo de elementos gráficos e todas as rodas estariam agrupadas nesse mesmo grupo.

Um possível resultado da aplicação da transformação encontra-se representado na figura 5.16.

```
Se existir rodas então
  Escreve o código necessário para criar o grupo
  Para cada entidade roda faz
    Remove o seu elemento gráfico correspondente do grupo geral
    Adiciona o elemento gráfico no grupo correspondente
  Fim do Para
Fim do Se
```

Figura 5.15: Pseudocódigo para produzir a configuração do editor gráfico

```
var palete = GmfTool!Palette.all.first();
var grupos = GmfTool!ToolGroup.all;
var grupoGeral = grupos.selectOne(r|r.title='Objects');

var grupoRodas = new GmfTool!ToolGroup;
grupoRodas.title = 'Rodas';
palete.tools.add(grupoRodas);

var roda = grupoGeral.tools.selectOne(r|r.title='Roda_Frente');
grupoGeral.tools.remove(roda);
grupoRodas.tools.add(roda);

roda = grupoGeral.tools.selectOne(r|r.title='Roda_Atras');
grupoGeral.tools.remove(roda);
grupoRodas.tools.add(roda);
```

Figura 5.16: Configuração do editor gráfico para DSL do robô de duas rodas

As regras de validação permitem validar as construções que o utilizador faz quando utiliza o editor gráfico. Para as regras serem aplicadas é necessário especificar qual o contexto onde se aplica cada uma das regras de validação.

O contexto é exatamente cada uma das rodas individualmente e as regras são escritas repetidamente para cada um dos contextos, no entanto as regras são iguais entre si.

A transformação responsável por produzir as regras de validação teria o comportamento descrito no pseudocódigo da figura 5.17.

Um possível resultado da aplicação da transformação encontra-se representado na figura 5.18. As regras de validação estão especificadas usando a linguagem *Epsilon Validation Language* (EVL).

```
Para cada entidade roda faz
  Escreve o contexto onde se aplica as regras
  Escreve as regras genéricas
Fim do Para
```

Figura 5.17: Pseudocódigo para produzir as regras de validação

```
context Roda_Frente {
  constraint PotenciaPositiva {
    check {
      return self.potencia.asInteger() > 0;
    }
    message {
      return 'Não é permitido potência negativa';
    }
  }
}

context Roda_Atras {
  constraint PotenciaPositiva {
    check {
      return self.potencia.asInteger() > 0;
    }
    message {
      return 'Não é permitido potência negativa';
    }
  }
}
```

Figura 5.18: Regras de validação para DSL do robô de duas rodas

A transformação responsável por produzir o artefacto *template* gerador de XML teria o comportamento descrito no pseudocódigo da figura 5.19. Neste caso, o próprio artefacto produzido é também uma transformação modelo-para-texto que utiliza os elementos do modelo gerado pela utilização do editor gráfico para produzir o XML que será carregado para o robô.

O artefacto gerado, percorre as entidades do modelo a partir da entidade *Inicio*, navegando pelas transições até à entidade seguinte. Quando esta é atingida invoca a operação *obterXML()* respeitante a essa entidade (contexto da operação) que lhe retorna o XML corretamente construído. Cada fração de XML é exposta sequencialmente no ficheiro de texto alvo, tratando-se desta ordem, a responsável por definir a sequência de operações a ser executada pelo robô.

O XML produzido para cada entidade roda é <*ETIQUETA nome="..."potencia="..."*/>, modificando-se a etiqueta para a entidade correta. Um possível resultado da aplicação

da transformação encontra-se representado na figura 5.20. O artefacto está especificado usando a linguagem *Epsilon Generation Language* (EGL).

```
Para cada entidade roda faz
  Escreve o contexto onde se aplica a operação
  Escreve o código do template alterando o nome da tag do XML
Fim do Para
```

Figura 5.19: Pseudocódigo para produzir o *template* gerador de XML

```
[%operation Roda_Frente obterXML() {
  return '<Roda_Frente nome="' + self.nome +
  '" potencia="' + self.potencia + '"/>';
} %]

[%operation Roda_Atras obterXML() {
  return '<Roda_Atras nome="' + self.nome +
  '" potencia="' + self.potencia + '"/>';
} %]
```

Figura 5.20: *Template* gerador de XML para DSL do robô de duas rodas

5.3.4 Considerações

Como é possível observar nas secções anteriores a abordagem aplicada nesta dissertação é facilmente utilizada noutros domínios distintos. Neste caso, sem um esforço excessivamente elevado seria possível desenvolver um gerador de linguagens para a família de DSLs do robô, ou seja, um possível fabricante que deseje fabricar robôs com um número indeterminado de rodas, bem como, com diferentes disposições das rodas no corpo do mesmo, não seria obrigado a desenvolver uma DSL de raiz para cada robô.

Observando esta solução do ponto de vista empresarial, permite reduzir custos no desenvolvimento de novas linguagens para cada robô. Do ponto de vista de mercado, permite colocar rapidamente um determinado produto em comercialização, fornecendo assim uma vantagem competitiva.

Esta solução implica um custo de desenvolvimento inicial mais elevado, bem como uma estratégia a longo prazo referente às possíveis evoluções do *hardware* do robô. Noutra perspetiva, permite a amortização do custo inicial do desenvolvimento do gerador de DSLs ao longo das diversas versões do robô que serão comercializadas posteriormente.

De forma genérica a solução apresentada, faz uso de mecanismos de herança e especificação, ou seja, desde que a entidade rodas no metamodelo (figuras 5.11 e 5.13) possa ser especificada num determinado tipo de roda desejada, esta não irá afetar o modelo de interação da DSL, então esta técnica pode ser usada.

5.4 Aplicação da Solução (Domínio IMA)

Nesta secção clarificam-se diversos detalhes relacionados com a solução adotada. Adicionalmente, evidenciam-se as diferenças nas configurações das partições *aviónicas* IMA que afetam significativamente cada uma das linguagens da família de DSLs.

5.4.1 Configurações do Modulo IMA

Considerando como base o *feature model* do domínio nas figuras 4.1 e A.2, verifica-se que quando se pretende desenvolver *software aviónico* para executar em partições distintas é necessário lidar com diferentes configurações relativas a cada partição.

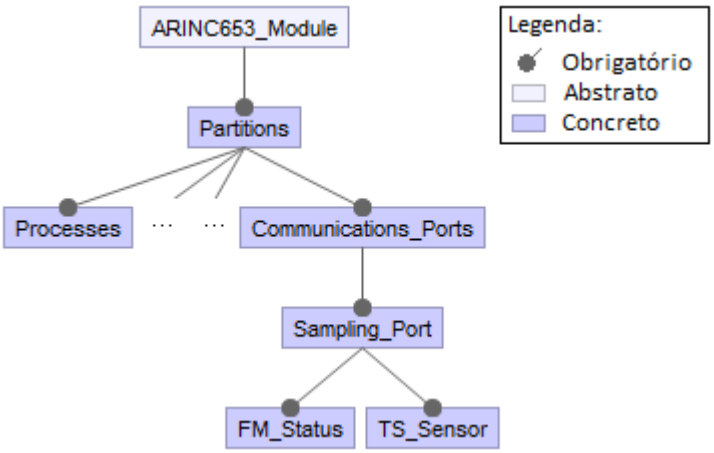


Figura 5.21: *Feature model* da partição *Flight Management*

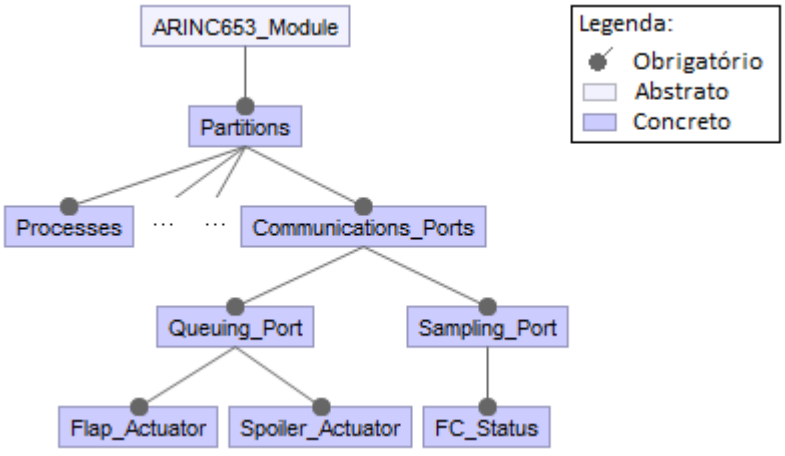


Figura 5.22: *Feature model* da partição *Flight Controls*

Supondo duas partições diferentes, denominadas *Flight Management* e *Flight Controls*, estando cada uma das configurações caracterizada pelos *feature model* das figuras 5.21 e 5.22 respetivamente. Verifica-se que cada partição tem uma configuração diferente relativamente aos portos de comunicação entre partições.

A partição *Flight Management* tem duas *Sampling Ports* e nenhuma *Queuing Port*. A partição *Flight Controls* tem duas *Queuing Ports* e uma *Sampling Port*. Logo a cardinalidade relativa a cada tipo de porto de comunicação entre partições é relevante. Utilizando uma abordagem de representação da cardinalidade descrita em [29], o novo *feature model* parcial do domínio pode ser observado na figura 5.23.

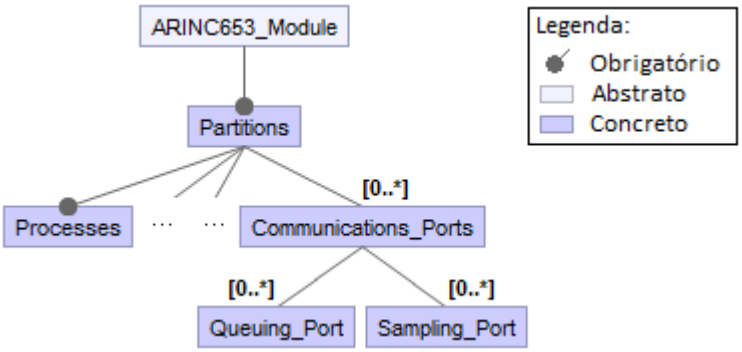


Figura 5.23: *Feature model* com cardinalidade

5.4.2 Família de DSLs para IMA

Em harmonia com a análise de domínio efetuada no capítulo 4, verifica-se que as principais variações do domínio centram-se nos portos de comunicação entre partições e nos mecanismos de comunicação e de sincronização entre processos. Estas variações, são relevantes e foram consideradas no desenvolvimento da solução.

Do ponto de vista estrutural das linguagens, as variações do domínio foram consideradas de forma diferente. Os mecanismos de comunicação e de sincronização entre processos fazem parte integral e estão constantemente presentes no metamodelo de todas as linguagens geradas. Os portos de comunicação entre partições, podem ou não estar presentes no metamodelo de cada DSL, ou seja, trata-se da variação do domínio (configurações diferentes) que origina a família de DSLs.

Utilizou-se o gerador de família de DSLs para produzir as duas linguagens referente às duas partições *Flight Management* e *Flight Controls*. De seguida explicitam-se as diferenças nos editores gráficos e nos restantes artefactos. Como complemento, demonstra-se através de pseudocódigo as operações necessárias para gerar esse artefactos.

5.4.2.1 Editor Gráfico das DSLs

A figura 5.24 apresenta o metamodelo dos conceitos de configuração da DSL para a partição *Flight Management*. A principal influência desse metamodelo no editor gráfico da DSL apresenta-se na figura 5.25.

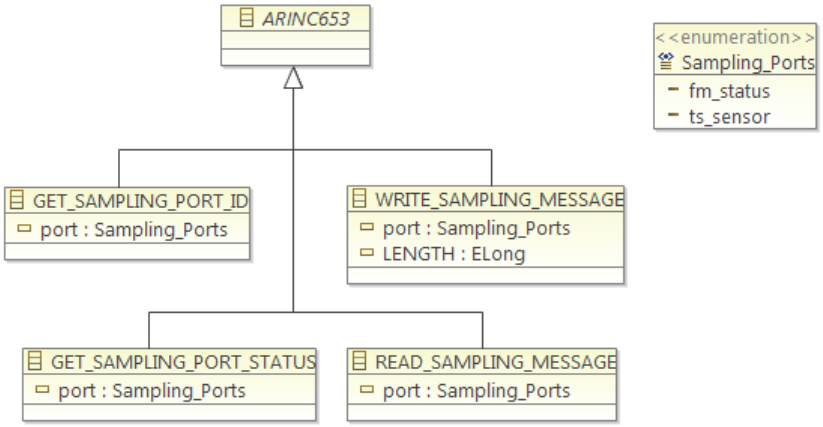


Figura 5.24: Metamodelo dos conceitos de configuração da DSL para a partição *Flight Management*

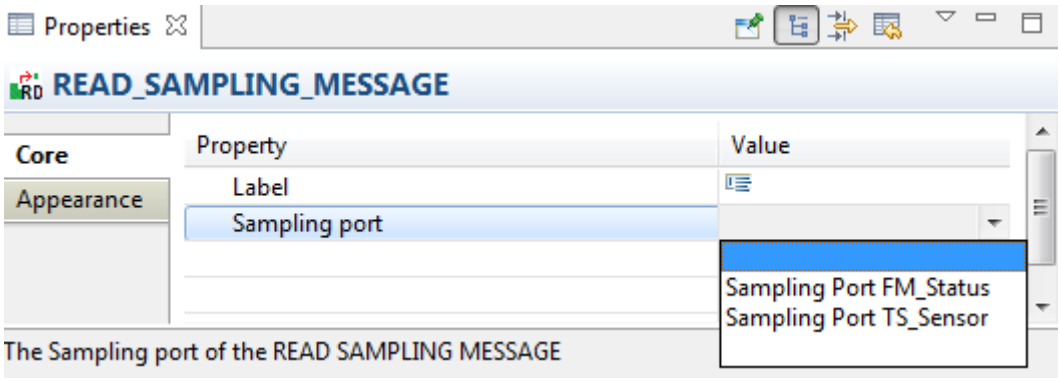


Figura 5.25: Aba de propriedades de *READ SAMPLING MESSAGE*

A entidade *ARINC653* na figura 5.24 é o ponto previamente definido onde é realizada a fusão com o metamodelo dos conceitos comuns do domínio apresentado na figura A.3.

As restantes entidades *GET SAMPLING PORT ID*, *GET SAMPLING PORT STATUS*, *READ SAMPLING MESSAGE* e *WRITE SAMPLING MESSAGE* estão posteriormente disponíveis como elementos gráficos no editor. Esta linguagem é composta por esses elementos, porque na configuração da partição *Flight Management* estão presentes duas *Sampling Ports*, caso contrário, estes elementos não estariam disponíveis para serem utilizados no editor gráfico.

Na figura 5.25 demonstra-se o fator mais significante que culminou com a solução implementada, trata-se da lista pendente que permite escolher a *Sampling Port*. Esta lista é fixa e limita a expressividade da linguagem de acordo com o requisito exigido e descrito na secção 5.1.

Devido à impossibilidade do utilizador adicionar e/ou remover portos de comunicação, evita-se que este especifique operações que não podem ser efetuadas, considerando a configuração da partição *aviónica*.

A figura 5.26 apresenta o metamodelo dos conceitos de configuração da DSL para a partição *Flight Controls*. A principal influencia no editor gráfico da DSL apresenta-se na figura 5.27.

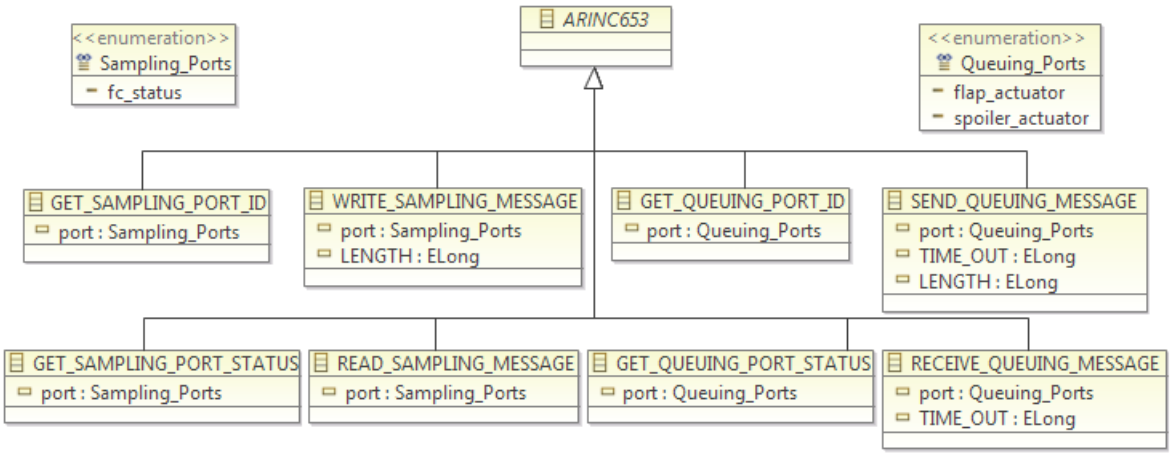


Figura 5.26: Metamodelo dos conceitos de configuração da DSL para a partição *Flight Controls*

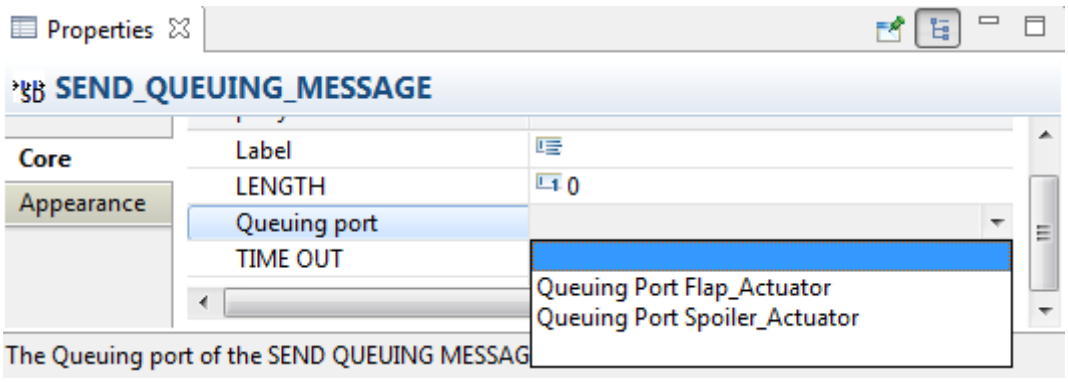


Figura 5.27: Aba de propriedades de *SEND QUEUING MESSAGE*

No metamodelo da figura 5.26 estão presentes as entidades *GET QUEUING PORT ID*, *GET QUEUING PORT STATUS*, *RECEIVE QUEUING MESSAGE* e *SEND QUEUING MESSAGE*. Estas entidades surgem porque a partição *Flight Controls* tem duas *Queuing Ports*. De forma idêntica à linguagem para a partição *Flight Management*, esta DSL também tem uma *Sampling Port*, significando isto, que estão também disponíveis as operações para esse tipo de porto de comunicação.

Nas linguagens geradas podem existir operações do standard ARINC 653 (ver na secção 2.1.1) que incidem nas *Queuing Ports* e *Sampling Ports*, se e só se, estas estiverem presentes na configuração da partição. Esta opção de desenho, permite reforçar o cumprimento do requisito que exige a limitação da expressividade da DSL. Deste modo evita-se que o utilizador tente gerar código (artefacto final pretendido) para operações que a partição não pode efetuar.

Adicionalmente, como efeito secundário, evita-se sobrecarregar a interface gráfica com elementos desnecessários que não podem ser utilizados.

5.4.2.2 Geração dos Artefactos

Nesta secção expõe-se o comportamento das transformações responsáveis por gerar a configuração do editor gráfico, as regras de validação e o *template* gerador de código C.

Para produzir estes artefactos é necessário saber quais são as entidades específicas da configuração. Essas entidades são conhecidas e já foram obtidas anteriormente para gerar o metamodelo dos conceitos de configuração.

Particularmente o artefacto *template* gerador de código C, necessita de diversa informação exterior para funcionar corretamente. Essa informação é obtida a partir do ficheiro XML de configuração do módulo no momento da produção do código C.

Todas as transformações são modelo-para-texto e a informação de saída é escrita em ficheiros separados.

O editor gráfico disponibilizado pelo Eclipse Epsilon, permite personalizar o aspeto gráfico, para tal é necessário discriminar essas configurações usando a linguagem EOL.

A transformação responsável por produzir a configuração do editor apresenta o comportamento descrito no pseudocódigo da figura 5.28. Genericamente, colocam-se os elementos gráficos no grupo correto da palete, ou seja, cria-se num novo grupo que aparecerá na barra de elementos gráficos normalmente visível no lado direito da janela do editor.

Uma fração do resultado da transformação encontra-se representado na figura 5.29.

```
1  Se existir portos de comunicação então
2    Escreve o código necessário para criar o grupo*
3    Para cada elemento gráfico relacionado com sampling ports faz
4      Remove o elemento do grupo geral
5      Adiciona o elemento ao grupo*
6    Fim do Para
7    Para cada elemento gráfico relacionado com queuing ports faz
8      Remove o elemento do grupo geral
9      Adiciona o elemento ao grupo*
10   Fim do Para
11 Fim do Se
```

Figura 5.28: Pseudocódigo para produzir a configuração do editor gráfico

(Legenda: grupo* - *Interpartition Communication*.)

```
var toolGroupPorts = new GmfTool!ToolGroup;
toolGroupPorts.title = 'Interpartition Communication';
toolGroupPorts.collapsible = true;
palette.tools.add(toolGroupPorts);

var toolGroupObjects =
  GmfTool!ToolGroup.all.selectOne(r|r.title = 'Objects');

var toolEntry;

toolEntry = toolGroupObjects.tools.selectOne(r|r.title =
  'GET_SAMPLING_PORT_ID');
toolGroupObjects.tools.remove(toolEntry);
toolGroupPorts.tools.add(toolEntry);

toolEntry = toolGroupObjects.tools.selectOne(r|r.title =
  'GET_SAMPLING_PORT_STATUS');
toolGroupObjects.tools.remove(toolEntry);
toolGroupPorts.tools.add(toolEntry);
```

Figura 5.29: Configuração do editor gráfico da DSL para a partição *Flight Management*

As regras de validação para serem aplicadas, é necessário especificar qual o contexto onde se aplica cada uma das regras. Neste domínio, o contexto é cada uma das operações especificadas no standard ARINC 653. Particularmente, as variações nas regras de validação referem-se às operações que incidem nas *Queuing Ports* e *Sampling Ports*, ou seja, caso não existam estas operações (como anteriormente referido, porque não existem *Queuing Ports* ou *Sampling Ports* na configuração da partição), as regras de validação para essas operações não são geradas.

A transformação responsável por produzir as regras de validação têm o comportamento descrito no pseudocódigo da figura 5.30.

Uma parcela do resultado da transformação encontra-se representada na figura 5.31. As regras de validação estão especificadas usando a linguagem EVL.

```
Se existir sampling ports então
  Escreve as regras de validação para cada contexto*1
Fim do Se
Se existir queuing ports então
  Escreve as regras de validação para cada contexto*2
Fim do Se
```

Figura 5.30: Pseudocódigo para produzir as regras de validação

(Legenda: contexto*1 - *GET SAMPLING PORT ID*, *GET SAMPLING PORT STATUS*, *READ SAMPLING MESSAGE* e *WRITE SAMPLING MESSAGE*; contexto*2 - *GET QUEUING*

PORT ID, *GET QUEUING PORT STATUS*, *RECEIVE QUEUING MESSAGE* e *SEND QUEUING MESSAGE*.)

```
context WRITE_SAMPLING_MESSAGE {
  constraint HasSamplingPort {
    check {
      return self.port.isDefined();
    }
    message {
      return 'SAMPLING PORT is not defined';
    }
  }
  critique HasLength {
    check {
      return self.LENGTH.asReal() <> 0;
    }
    message {
      return 'MESSAGE LENGTH is set to 0';
    }
  }
}
```

Figura 5.31: Regras de validação da DSL para a partição *Flight Management*

A transformação responsável por produzir o artefacto *template* gerador de código C apresenta o comportamento descrito no pseudocódigo da figura 5.32. O artefacto produzido é uma transformação modelo-para-texto que utiliza os elementos do modelo gerado pela utilização do editor gráfico para produzir o código.

Este *template* produzido, processa os elementos do modelo de forma específica e apropriada. Quando é necessário gerar o código de uma dada entidade (contexto da operação), é então invocado a operação *code()*.

Uma fração do resultado da transformação apresentada na figura 5.32 encontra-se demonstrado na figura 5.33. O artefacto está especificado usando a linguagem EGL.

```
Se existir sampling ports então
  Escreve a operação code() para cada contexto*1
Fim do Se
Se existir queuing ports então
  Escreve a operação code() para cada contexto*2
Fim do Se
```

Figura 5.32: Pseudocódigo para produzir o *template* gerador de código C

(Legenda: contexto*1 - *GET SAMPLING PORT ID*, *GET SAMPLING PORT STATUS*, *READ SAMPLING MESSAGE* e *WRITE SAMPLING MESSAGE*; contexto*2 - *GET QUEUING*

PORT ID, *GET QUEUING PORT STATUS*, *RECEIVE QUEUING MESSAGE* e *SEND QUEUING MESSAGE*.)

```
1 [%operation WRITE_SAMPLING_MESSAGE code( ... ) {
2   var samplingPortName = ... ;%]
3 WRITE_SAMPLING_MESSAGE( ... ,&[%=samplingPortName%]_ReturnCode);
4 if ([%=samplingPortName%]_ReturnCode != NO_ERROR) {
5   printf("Couldn't write the sampling message: %d", ... );
6 }
7 [%}%]
```

Figura 5.33: *Template* gerador de código C da DSL para a partição *Flight Management*

5.4.3 Considerações

A forma de aplicação da solução no domínio IMA e no domínio do robô são idênticas, no entanto dependendo do objetivo da família de DSLs, existem discrepâncias na forma de produzir os artefactos.

Genericamente, o robô pode conter um número ilimitado de elementos novos no editor gráfico. Este facto obriga que as transformações sejam forçadas a lidar com essa característica.

No domínio IMA, tal não acontece, os elementos gráficos são limitados às operações definidas no standard ARINC 653, o que facilita a geração da configuração do editor gráfico, das regras de validação e do *template* gerador de código C. Neste caso, só é necessário lidar com a presença ou não das operações que incidem sobre as *Sampling Ports* ou *Queuing Ports*.

Noutra perspetiva, em IMA é necessário lidar com os portos de comunicação e com a configuração relacionada com estes. Essa informação é externa à linguagem desenvolvida, logo é necessário garantir que os artefactos gerados têm acesso a esses dados. Este fator é especialmente crítico no *template* gerador de código C, porque este artefacto é ele próprio uma transformação e é necessário garantir que essa transformação quando for executada, também terá acesso às informações externas quando necessário.

De um modo geral, dependendo do domínio onde a solução é aplicada, poderá ser mais complexo desenvolver as transformações que geram os artefactos, e/ou gerir o acesso à informação externa de forma a que as transformações funcionem corretamente.

A abordagem adotada como solução nesta dissertação, cumpre todos os requisitos exigidos para o domínio IMA, em especial, permite limitar a expressividade das linguagens produzidas. Este requisito, impõe uma limitação à liberdade do utilizador face à configuração de uma partição *aviónica* de um módulo, tornando-se essencial para prevenir erros no *software* desenvolvido. No entanto, esta solução apresenta uma característica menos favorável relacionada com o desempenho da geração de uma nova linguagem da

família de DSLs. Esta característica encontra-se abordada no capítulo 6.

6

Implementação

No decorrer da presente dissertação implementaram-se dois protótipos. O primeiro operacionaliza o conceito de família de DSLs. O segundo (versão empresarial) é um refinamento do primeiro protótipo com o objetivo de fornecer uma linha orientadora para um produto comercial, bem como resolver dificuldades de desempenho consequentes das necessidades dessa linha orientadora.

O problema de desempenho, surge devido a elementos externos à solução implementada, ou seja, não estão relacionados com esta dissertação. O conceito de geração automática de DSLs, que também pode ser entendido como fábrica de linguagens é provado no protótipo da secção 6.1. A necessidade de ir mais além nesta dissertação, construindo-se um produto pré-comercial colide com questões tecnológicas ainda em investigação, o que conduziu ao desenvolvimento do protótipo da secção 6.2.

A geração automática de uma DSL usando como base o metamodelo da linguagem, e com recurso a ferramentas disponíveis nas *Languages Metamodeling Workbench* (LMW) é um processo significativamente lento. Este tipo de ferramentas são compostas por um conjunto de transformações que produzem automaticamente o código completo dos editores gráficos. Esta geração é lenta e está dependente da implementação interna das transformações nas ferramentas.

Este problema de desempenho das transformações no paradigma de desenvolvimento MDD, foi aprofundado em outras dissertações como é exemplo em [17], mas ainda se encontra na fase de transição para aplicação nas ferramentas atualmente existentes. Visto ser necessário aprofundar os procedimentos e metodologias com o intuito de acelerar a execução de transformações nas LMW, o segundo protótipo, segue um paradigma diferente. Trata-se de uma única DSL configurável.

O foco desta dissertação é o protótipo da secção 6.1. O protótipo empresarial é uma solução para contornar as dificuldades de desempenho e não se trata de uma alternativa que satisfaça todos os requisitos inicialmente exigidos (ver na secção 6.2.2).

Nos anexos, em A.14, A.15, A.16, A.17 e A.18 encontra-se o manual de utilizador dos

protótipos. Adicionalmente em A.19 e A.20 encontra-se o exemplo de uma aplicação *aviónica* desenvolvida com recurso ao protótipo empresarial.

6.1 Protótipo

O desenvolvimento do protótipo, efetuou-se com recurso a diversas tecnologias. Algumas amplamente conhecidas como JAVA, XML e ANT, outras mais utilizadas na abordagem MDD como EOL, EVL e EGL. Adicionalmente, foi necessário desenvolver *plug-ins* para o Eclipse.

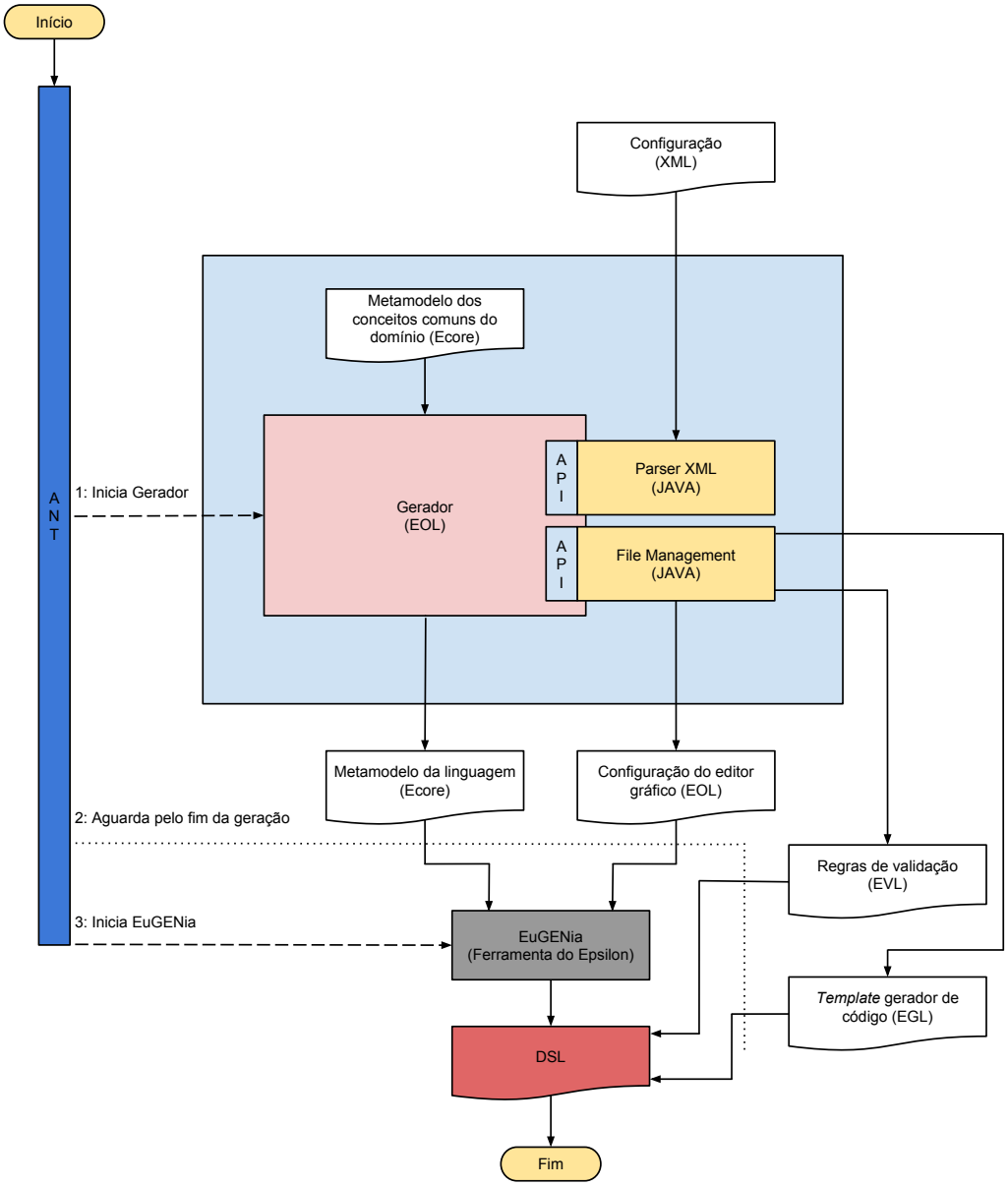


Figura 6.1: Processo de funcionamento e comunicação dos componentes da solução

Na figura 6.1 ilustram-se os componentes, o fluxo de informação e a dinâmica de funcionamento entre cada componente individual.

O utilizador quando pretende tirar partido de uma nova linguagem, executa a solução. Posteriormente escolhe o ficheiro XML que contém a configuração do módulo, bem como seleciona a partição *aviónica*. Finalmente resulta a DSL completamente gerada e pronta a ser utilizada.

Para a sequência de passos ser amiga do utilizador, requereu-se uma orquestração interna do processo que é efetuada por um *script* ANT. No primeiro passo, o gerador da família de DSLs, solicita através de uma API do *Parser* XML, a informação necessária para produzir os artefactos.

Alguns destes artefactos são guardados através da utilização de um módulo externo ao gerador que fornece uma API própria para escrever a informação em ficheiros. No caso do metamodelo da linguagem é armazenado num ficheiro diretamente pelo gerador.

Quando termina a produção do metamodelo da linguagem, a configuração do editor gráfico, as regras de validação e o *template* gerador de código C, o ANT invoca a execução do EuGENia que produz o editor gráfico da linguagem terminando assim o processo de geração da mesma. A DSL fica de seguida disponível para utilização.

Tratando-se de uma dissertação empresarial, considerou-se a utilização mínima de diferentes tecnologias e linguagens. Desta forma evita-se um elevado número de camadas intermédias de *software* para fazer a ponte entre essas mesmas tecnologias. Assim, sempre que possível, foi utilizada a linguagem de programação EOL.

6.1.1 Orquestração do Processo

Com o intuito de automatizar o processo de geração das linguagens, desenvolveu-se um *script* ANT. Este permite executar de forma sequencial, primeiro o gerador e posteriormente o EuGENia. Estas tarefas aparecem identificadas na figura 6.1.

```
1  <?xml version="1.0"?>
2  <project default="main">
3    <target name="loadModels">
4      <epsilon.emf.loadModel name="S1" ... />
5      ...
6      <epsilon.emf.loadModel name="T" ... />
7    </target>
8    <target name="main" depends="loadModels">
9      <epsilon.eol src="./AuxCode/Main.eol">
10       <model ref="S1"/>
11       ...
12       <model ref="T"/>
13     </epsilon.eol>
14     <epsilon.eugenia src="./Artefacts/imadsl.ecore"/>
15   </target>
16 </project>
```

Figura 6.2: *Script* ANT da solução

O ANT utiliza XML para descrever a automatização dos processos, bem como as suas dependências. Na figura 6.2 é demonstrado parcialmente o *script* da solução. Da linha 3 à 7 é instruído para carregar os modelos em memória. Da linha 9 à 13 requer-se a execução do gerador com os modelos previamente carregados. Na linha 14 solicita-se a execução do EuGENia.

6.1.2 Acesso a Informação Exterior (*Parser* XML)

Para gerar uma linguagem da família de DSLs é necessário obter a configuração da partição que se encontra num ficheiro XML. Essa informação está especificada de acordo com o *schema* presente no standard ARINC 653.

Desenvolveu-se um *plug-in* com o objetivo de fornecer funcionalidades adicionais que não estão presentes no Eclipse Epsilon. De forma complementar, esta estratégia permite separar convenientemente o módulo gerador de DSLs, da gestão de informação com origem ou destino no exterior.

O *plug-in* foi desenvolvido com recurso à linguagem JAVA e exporta duas APIs. A primeira permite obter a informação necessária proveniente do ficheiro com a configuração e gere diretamente o *Parser* XML. A segunda API permite gravar informação em ficheiros de texto de forma simplificada e encontra-se explicado na secção 6.1.3.

Para aceder à informação do ficheiro é necessário indicar a localização do mesmo no sistema de ficheiros local, bem como a partição *aviónica* correspondente aos dados que se pretendem obter. Posteriormente, a informação é internamente processada e fica disponível para ser consultada pelo gerador de DSLs através da API.

Existem duas formas de iniciar o *Parser* XML. Pode-se solicitar que o utilizador indique a localização do ficheiro através de caixas de diálogo ou indicar diretamente a localização completa do ficheiro em forma de texto (*path*).

Estas duas alternativas prendem-se com a necessidade de acautelar o acesso à informação em situações distintas. A primeira é quando se pretende gerar a DSL, nesse momento é necessário saber pela primeira vez a localização do ficheiro que só o utilizador conhece. A segunda é quando a linguagem é executada e as regras de validação ou o *template* gerador de código C necessitam de aceder a informação externa para cumprirem a sua função, nesse momento já conhecem a *path* porque ficou internamente guardada nos artefactos quando a DSL foi produzida.

Na figura 6.3 apresenta-se a API do *Parser* XML. Das rotinas apresentadas, as mais relevantes são o *loadFile()* que permite iniciar o *Parser* apresentando ao utilizador os diálogos para escolha da localização do ficheiro. O *loadFile(String , int)* que permite iniciar o *Parser* quando já se conhece a *path* e a partição. O *getSamplingPorts()* e *getQueuingPorts()* para obter a configuração de todas as *Sampling Ports* e *Queuing Ports* respetivamente.

Durante o desenvolvimento, teve-se a preocupação em consumir poucos recursos que

de alguma forma pudessem influenciar o desempenho da solução, sendo assim, foi implementado um SAX[1] *Parser* que permite consumir pouca memória, pois, neste caso só são instanciados objetos relacionados com os portos de comunicação.

```
void loadFile()
void loadFile(String filePath, int nrOfPartition)
void unloadFile()
void setFilePath(String filePath)
String getFilePath()
void setNrOfPartition(int nrOfPartition)
int getNrOfPartition()
List<Hashtable<String, String>> getSamplingPorts()
List<Hashtable<String, String>> getQueuingPorts()
```

Figura 6.3: API do *Parser* XML

6.1.3 Gravação de Artefactos (*File Management*)

O *plug-in* Eclipse que fornece a API mencionada na secção 6.1.2, também disponibiliza uma API que permite guardar informação em ficheiros de texto denominada *File Management*.

Esta API foi desenvolvida devido a dois fatores. O primeiro refere-se ao artefacto *template* gerador de código C, que é ele próprio composto por código EGL. Visto que o Epsilon só fornece o *template* EGL como transformação modelo-para-texto e que simultaneamente permite guardar em ficheiros externos, então desenvolver um *template* EGL para produzir EGL seria de elevada complexidade. Esta situação poderia originar que o meta-código responsável por gerar o código alvo, interpretasse o código alvo como seu próprio código. Este motivo também reforçou a pretensão de utilizar sempre que possível a linguagem EOL no protótipo.

O segundo fator refere-se à simplicidade da tarefa exigida, pois só é necessário anexar texto de forma sequencial no final do ficheiro. Logo, não é necessário mais além, do que efetuar *append* de código EGL pertencente ao *template* gerador de código C, num ficheiro.

De forma complementar, foram adicionadas operações que permitem efetuar tarefas com os ficheiros tais como, apagar, copiar ou limpar o seu conteúdo. Estas operações são utilizadas pelo módulo gerador de DSLs.

Na figura 6.3 apresenta-se a API do *File Management*. Das rotinas apresentadas, as mais relevantes são o *open(String)* que permite abrir em modo de escrita o ficheiro onde será guardada a informação. O *appendLine(String)* que permite escrever dados no ficheiro previamente aberto. Finalmente, o *close()* que permite terminar o acesso ao ficheiro.

[1] http://www.saxproject.org/ e http://docs.oracle.com/javase/tutorial/jaxp/sax/

```
boolean open(String path)
boolean appendLine(String data)
boolean close()
void copyFile(String source, String target)
void deleteFile(String path)
void cleanUpFile(String path)
```

Figura 6.4: API do *File Management*

6.1.4 Módulo Gerador de DSLs

Implementou-se o módulo gerador de DSLs com recurso a uma só linguagem designada por EOL. No entanto este módulo tem de interagir com ficheiros representativos do formalismo Ecore, bem como produzir artefactos em linguagens como o EOL, EVL e EGL (figura 6.1).

Manipular a informação proveniente da API do *Parser* XML é totalmente transparente para o gerador, pois recebe a informação dos portos de comunicação em formato texto e armazenado numa estrutura do tipo dicionário (*hash table*), que pode consultar sempre que é necessário.

De igual forma, manipular a informação que é guardada em cada um dos ficheiros correspondente aos artefactos gerados, também é transparente para o gerador, pois este interpreta o EOL, EVL e EGL de cada um dos artefactos como se tratasse de texto, sem qualquer significado adicional. Desta forma, lidar com a API do *File Management* é do ponto de vista do gerador, como armazenar texto num ficheiro.

Produzir o metamodelo da linguagem é uma importante tarefa do gerador. Este utiliza o metamodelo dos conceitos comuns do domínio que foi desenvolvido anteriormente de forma manual. Produz automaticamente o metamodelo dos conceitos de configuração e finalmente faz a fusão dos dois metamodelos anteriores.

O metamodelo dos conceitos comuns do domínio é mantido como *template*, por outras palavras, sempre que se produz uma nova DSL, este metamodelo é copiado para outra localização diferente de onde se encontra originalmente. Essa cópia será então fundida com o metamodelo dos conceitos de configuração. Deste modo, mantém-se inalterado o metamodelo dos conceitos comuns do domínio para gerar uma nova linguagem quando pretendido.

A operação de fusão é ilustrada na figura 6.5. Em primeiro lugar o gerador identifica quais as entidades que têm o mesmo nome em ambos os metamodelos. Todas as referências e atributos em cada entidade identificada, são copiados da entidade do metamodelo dos conceitos de configuração para a entidade do metamodelo dos conceitos comuns do domínio. Todas as outras entidades são integralmente copiadas do metamodelo dos conceitos de configuração para o metamodelo dos conceitos comuns do domínio. Devido à

representação interna do formalismo Ecore, estas duas operações garantem a fusão dos dois metamodelos parciais, formando assim o metamodelo da linguagem.

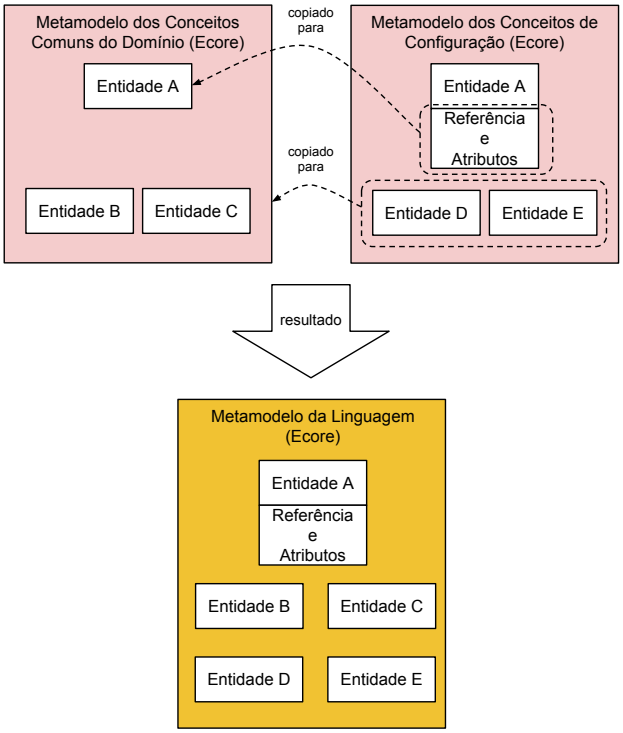


Figura 6.5: Fusão de metamodelos

Lidar com a geração dos artefactos já foi globalmente demonstrado na secção 5.4.2.2, mas é necessário neste momento considerar, que do ponto de vista do desenvolvimento os pseudocódigos aí especificados foram implementados com recurso à linguagem EOL.

A configuração do editor gráfico é guardada num ficheiro com o nome *ECore2GMF.eol* e na mesma localização do metamodelo da linguagem. Estes dois fatores são importantes, pois só desta forma o EuGENia tem em consideração as configurações no momento de gerar os editores gráficos da linguagem.

6.1.5 Considerações

Não obstante de se tratar de um protótipo com intuito de provar o conceito de fábrica de DSLs, foi também tido em consideração o desempenho da solução. Esse requisito de qualidade é importante para a indústria, logo em algumas decisões tomadas no desenvolvimento do protótipo (decisões que não afetam a prova do conceito) foi considerada a decisão menos onerosa do ponto de vista do desempenho da solução.

Essas decisões são principalmente a uniformidade da linguagem usada no gerador de DSLs, bem como a utilização de um *Parser* XML mais eficiente e externo ao Epsilon. É possível utilizar os mecanismos da LMW para manipular o XML que contêm as configurações do módulo *aviónico*, no entanto, esses mecanismos carregam em memória todo

o ficheiro documento (DOM). Esse processo traduz-se num maior consumo de memória, num maior consumo de tempo no processo de instanciação de todos os objetos, bem como a instanciação de informação que nunca será usada, logo desnecessária.

As opções tomadas foram suficientes para atingir um bom desempenho no módulo gerador, no entanto devido à menor eficiência da ferramenta EuGENia, a solução no global apresenta um desempenho menos adequado, o que levou ao desenvolvimento de um novo protótipo abordado na secção 6.2.

A implementação deste protótipo mostrou claramente que é necessário desenvolver um gerador específico para produzir cada família de DSLs, pois cada uma tem características ímpares que influenciam quais as entidades a manipular nos metamodelos, bem como a lógica de geração dos artefactos (por exemplo pode ser observado comparando o pseudocódigo da figuras 5.15 e 5.28).

Noutra perspetiva as alterações a efetuar no gerador de linguagens para ser aplicado a outros domínios, são em número razoável e bem localizados. Estes dois fatores favorecem a aplicação da solução noutras áreas.

O gerador de linguagens é capaz de fabricar e apresentar várias linguagens. Deste ponto de vista pode ser considerado um único produto que se molda a várias configurações. Por outro lado pode ser visto como uma fábrica de linguagens, que depois são separadas do gerador e tornam-se produtos independentes.

6.2 Protótipo Empresarial

A necessidade de ir mais além nesta dissertação, construindo-se a base de um produto completo com características mais refinadas, nomeadamente relacionado com o desempenho, proporcionou o desenvolvimento deste protótipo.

Nesta versão segue-se também uma abordagem de desenvolvimento de uma DSL com recurso a MDD como explicado nas secções 2.3.2 e 2.3.3 e também utilizado em [43].

Este protótipo não apresenta um novo metamodelo da linguagem construído de raiz. De facto, trata-se de um redesenho do metamodelo da linguagem apresentado no protótipo anterior. Concretamente, o metamodelo dos conceitos comuns do domínio manteve-se inalterado e foi estendido manualmente de forma a representar os novos conceitos pretendidos. Por outras palavras, neste protótipo não existem dois metamodelos parciais, existe sim o metamodelo completo da linguagem, cuja diferença face ao metamodelo dos conceitos comuns usado no protótipo da secção 6.1 é o fragmento do metamodelo da linguagem manualmente construído e apresentado na figura 6.6 (as entidades *Partition* e *ARINC653* na zona sombreada são os elementos de ligação do fragmento).

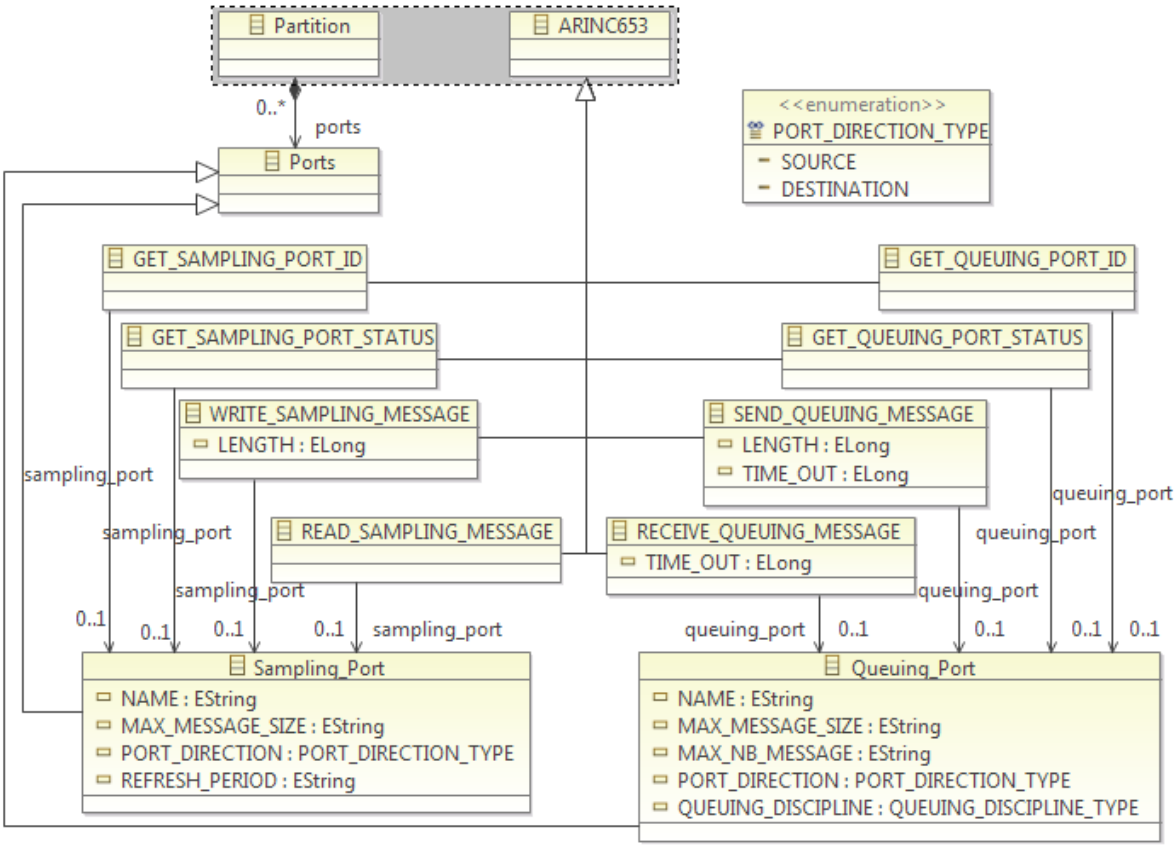


Figura 6.6: Fragmento do metamodelo da linguagem

Como se pode observar no fragmento, as *Queuing Ports* e *Sampling Ports* estão presentes permanentemente na linguagem. Desta forma, a linguagem contempla todas as variações possíveis do número e configuração dos portos de comunicação, pois é possível adicionar ou remover portos, bem como alterar a informação dos existentes.

Esta característica da nova linguagem, contraria um dos requisitos exigidos (ver na secção 5.1) que impõe a limitação à expressividade da linguagem para conter a liberdade do utilizador, tendo em conta a configuração da partição *aviónica* de um módulo.

Para contornar este problema, não se realizou a ligação entre as entidades *Sampling_Port* e *Queuing_Port* com a sintaxe concreta, ou seja, não foi feita nenhuma anotação no Ecore referente a estas entidades. De acordo com o comportamento esperado do EuGENia, na geração dos editores, este produz todos os elementos internos que garantem o funcionamento do editor gráfico, mas não produz os elementos gráficos correspondentes às *Queuing Ports* e *Sampling Ports*.

Desta forma, o utilizador não consegue adicionar, remover ou alterar os portos de comunicação. No entanto quando pretende usar alguma operação do standard ARINC 653 que necessite dos portos (por exemplo: *GET SAMPLING PORT ID*), não estará nenhum disponível para selecionar na lista pendente.

Noutra perspetiva, a ausência de portos de comunicação para selecionar, quer dizer

que a configuração do módulo *aviónico* ainda não foi carregada. A solução foi alterar o código JAVA do editor gráfico manualmente. Em primeiro lugar, quando a DSL é executada, é exigido que o utilizador selecione um ficheiro XML e posteriormente a partição *aviónica* (o *Parser* XML foi adaptado do protótipo anterior). Findo este processo, são adicionados às estruturas de dados internas do editor (mas não visíveis para manipular graficamente) todos os portos de comunicação contendo a configuração completa sobre os mesmos. Assim, ficam disponíveis para selecionar na lista pedente quando necessário.

Este paradigma de DSL configurável, apresenta uma vantagem relevante quando comparado com o anterior protótipo da secção 6.1. Trata-se de não ser necessário aceder a informação exterior para executar as regras de validação ou o *template* gerador de código C.

Esta situação verifica-se porque quando o utilizador usa o editor gráfico da linguagem, este produz um modelo que representa a informação presente no editor (mesmo que o elemento gráfico não esteja visível). De igual forma, quando se adiciona por código os portos de comunicação, o editor gráfico também adiciona essa informação ao modelo, sendo assim, qualquer artefacto que use esse modelo consegue aceder à informação lá presente para executar a sua função.

As regras de validação e o *template* gerador de código C, também foram reaproveitados do protótipo anterior. O procedimento foi simples, produziu-se uma linguagem com o gerador de DSLs, relativo a uma partição que tinha *Queuing Ports* e *Sampling Ports*. De seguida retiraram-se os artefactos para o novo protótipo. Por último adaptaram-se esses artefactos para aceder ao modelo produzido pelo editor gráfico de forma a poder obter a informação necessária do mesmo, ao invés de usar fontes externas como no protótipo anterior.

6.2.1 Eclipse RCP

A DSL configurável foi exportada como uma aplicação Eclipse RCP. Para o efeito foi preciso instruir o EuGENia que esse seria o objetivo final. Essa instrução foi feita através da anotação *rcp="true"* na entidade *Partition*, pois esta é a entidade que representa o objeto raiz do metamodelo. De seguida, executou-se o EuGENia que produziu o editor gráfico com as devidas adaptações para executar neste ambiente.

As regras de validação são executadas quando o utilizador clica na opção *Validate* no menu *Edit* da barra de menus. Mas não existe forma de executar o *template* gerador de código C quando se trata de uma aplicação RCP. Para o efeito, foi adicionado um botão na barra de ferramentas com o nome *Exportar para C*. Quando o botão é pressionado executa-se código JAVA. Este obtém o modelo produzido pela manipulação do editor gráfico e executa o *template* gerador de código C com esse modelo. O resultado é armazenado numa localização escolhida pelo utilizador.

A adição do botão na barra de ferramentas foi efetuada com recurso a pontos de

extensão disponibilizado pelo Eclipse. Para o efeito foi adicionado um *action set*[2] e uma *action*[3].

Por último criou-se um elemento no projeto eclipse denominado *Product Configuration* onde se especificou as dependências de outros *plug-ins* necessários para executar o editor gráfico, bem como se forneceu outras informações adicionais de configuração. Executou-se o assistente de exportação que produziu a DSL configurável RCP.

6.2.2 Considerações

O protótipo empresarial apresenta um paradigma de DSL configurável. Esta solução deve-se à configuração efetuada automaticamente no momento da inicialização da linguagem, e de posteriormente não ser permitido ao utilizador alterar essa configuração.

Uma importante diferença entre os dois protótipos prende-se com o fator relativo à expressividade da linguagem. O protótipo que prova o conceito de família de DSLs automaticamente geradas, produz linguagens que têm uma sintaxe abstrata restrita e desse modo limita-se a expressividade da DSL logo a partir da sua conceção. O protótipo industrial apresenta uma sintaxe abstrata mais ampla e limita essa expressividade por configuração e ocultação dos elementos que permitem modificar essa configuração.

Algumas cedências foram efetuadas neste protótipo. Por exemplo, mesmo que não exista *Sampling Ports* (o mesmo se verifica no caso de *Queuing Ports*) na configuração da partição *aviónica* do módulo escolhido, as operações do standard ARINC 653 relativas a *Sampling Ports* estarão na mesma disponíveis na barra de ferramentas, o que não se verifica no protótipo da secção 6.1. Seria possível lidar com a remoção dinâmica destes elementos gráficos, mas seria necessário alterar um elevado número de componentes do editor gráfico para comportar essa solução.

O aspeto gráfico e a usabilidade da DSL configurável é exatamente igual às linguagens produzidas com recurso ao gerador de DSLs, com exceção de duas diferenças do ponto de visto do utilizador. A primeira prende-se com a característica referida no parágrafo anterior. A segunda está relacionado com o facto da DSL configurável ser uma versão Eclipse RCP e o gerador de DSLs executar em ambiente de desenvolvimento.

[2] http://wiki.eclipse.org/FAQ_What_is_an_action_set%3F
[3] http://wiki.eclipse.org/FAQ_How_do_I_add_actions_to_the_global_toolbar%3F

7

Avaliação de Usabilidade e Validação Funcional

Sem uma avaliação de usabilidade às linguagens produzidas pelo gerador de DSLs, a presente dissertação estaria incompleta, nem seria possível provar que se cumpriu o segundo e terceiro requisito descritos na secção 5.1.

Como o gerador de linguagens é capaz de produzir um número de DSLs quase infinito (igual ao número de diferentes configurações que possam existir), efetuou-se esta avaliação com recurso a duas linguagens.

As linguagens avaliadas não foram produzidas pelo gerador de DSLs, mas sim pelo protótipo empresarial, ou seja, a DSL configurável, apresentada na secção 6.2. Este fator não compromete a credibilidade da avaliação, pois uma dada linguagem produzida pelo gerador de DSLs apresenta exatamente o mesmo aspeto gráfico, comportamento e resultados que a DSL configurável. Isto considerando-se a mesma configuração da partição aplicado em ambos os casos. Esta garantia é dada por construção, pois a DSL configurável foi desenvolvida à imagem do gerador de DSLs, reutilizando-se inclusive partes substanciais e centrais deste.

Existem unicamente, duas diferenças do ponto de visto do utilizador. A primeira prende-se com o facto da DSL configurável ser uma versão Eclipse RCP e o gerador de DSLs executar em ambiente de desenvolvimento, logo é necessário seguir procedimentos diferentes para iniciar a execução de cada um deles. Esta diferença não é relevante porque o teste de usabilidade foi iniciado com a DSL já em execução, mas antes de ter sido escolhido o ficheiro XML e a partição.

A segunda diferença, trata-se da permanência das operações do standard ARINC 653 para os portos de comunicação mesmo que estes não existam na configuração da partição *aviónica* escolhida. Esta diferença também não se considera relevante porque uma das duas linguagens testadas só usa *Sampling Ports* e a outra só usa *Queuing Ports*, não causando entropia na escolha das operações a ser usadas pelo utilizador.

Devido a existência de poucas linguagens gráficas para a aeronáutica e este apresentar-se como um domínio restrito e muito dispendioso, não foi possível confrontar as duas DSLs com outras linguagens como o SCADE Suite (ver na secção 2.1.5.3). Sendo assim, no sentido lato, o objetivo da avaliação da usabilidade foi comparar as linguagens com a linguagem C. Trata-se da principal linguagem utilizada no domínio, pois o código é produzido maioritariamente (com larga margem) por especialistas de forma manual.

Para efetuar esta avaliação prepararam-se slides, questionários, equipamentos informáticos e uma sala com condições confortáveis. Recrutaram-se utilizadores alvo de acordo com o perfil especificado na secção 4.5. Com exceção de um utilizador que não tem experiência no domínio, os restantes seis elementos têm experiência elevada ou razoável no desenvolvimento de *software aviónico*.

A usabilidade é considerada a medida em que um produto pode ser usado por utilizadores específicos para alcançar objetivos específicos com efetividade, eficiência e satisfação num contexto de uso especificado [23]. Portanto, avaliou-se principalmente a eficácia contando o número de erros efetuados no desenvolvimento. A eficiência medindo o tempo consumido e a satisfação através de questões apropriadas efetuadas aos utilizadores depois de terem realizado os testes de usabilidade.

Esta avaliação empírica da usabilidade tratou-se de uma adaptação do processo descrito na secção 2.3.4.2 e particularmente ilustrado na figura 2.15.

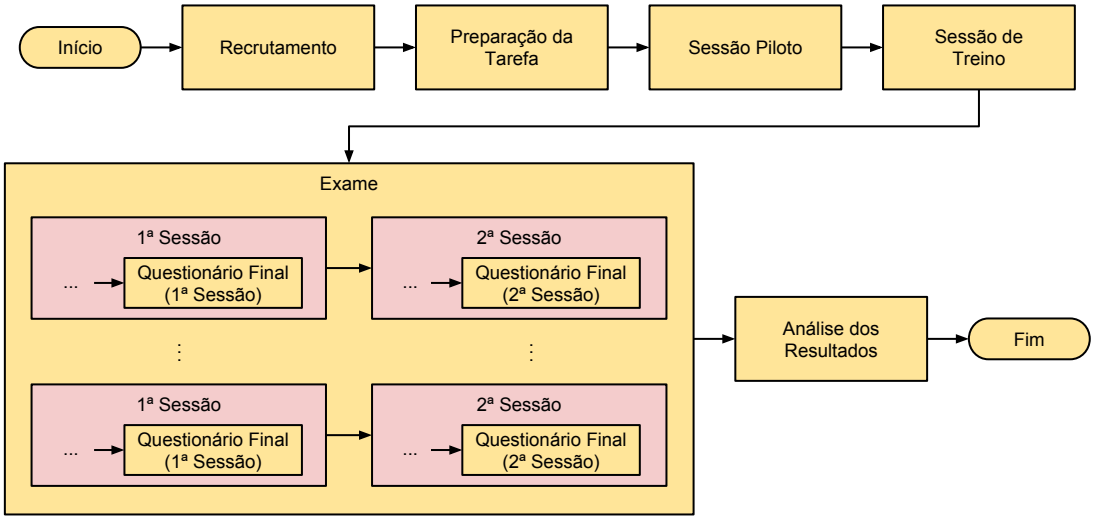


Figura 7.1: Processo de avaliação

Na figura 7.1 encontra-se ilustrado o processo de avaliação que foi executado no decorrer desta dissertação. A adaptação verifica-se no questionário final que não é uma etapa distinta do exame. Este facto ocorreu porque o exame tem duas sessões por cada grupo de utilizadores, sendo assim, necessitou-se de efetuar os questionários no final de cada sessão (ver na secção 7.2.5).

7.1 Objetivos

Com a avaliação de usabilidade desejou-se perceber até que ponto os objetivos foram cumpridos, para isso pretendeu-se responder às seguintes questões relacionadas com as linguagens produzidas pelo gerador de DSLs:

- Programar com a utilização das DSLs para auxiliar o desenvolvimento de aplicações *aviónicas* é mais eficaz do que usar só a linguagem C?
- É mais eficiente utilizar as DSLs para gerar numa primeira fase o esqueleto da aplicação do que usar só a linguagem C?
- Os utilizadores estão satisfeitos com a utilização das DSLs no processo de desenvolvimento?

No campo da satisfação, além da perspetiva global, é importante responder mais detalhadamente às duas questões seguintes:

- Os participantes estão mais confiantes na solução que permite gerar o esqueleto de código da aplicação, ao invés de usar só a linguagem C ?
- As DSLs facilitaram o processo de desenvolvimento?

7.2 Processo de Avaliação

O processo completo de avaliação compreende alguns procedimentos com o intuito de responder do melhor modo possível às questões que se apresentou na secção 7.1.

Nesta secção descrevem-se as primeiras cinco das seis etapas apresentadas na figura 7.1. Estas foram percorridas sequencialmente e aplicadas de forma adaptada ao domínio, bem como às linguagens avaliadas. De forma complementar, foram utilizados em diversas ocasiões os materiais referidos na secção 7.2.2. A última etapa é a análise dos resultados, e devido à sua especial relevância apresentou-se separadamente na secção 7.3.

7.2.1 Recrutamento

O primeiro procedimento prendeu-se com a correta seleção dos utilizadores para efetuar a avaliação de usabilidade.

De um conjunto de colaboradores da empresa, selecionou-se um grupo alargado com características semelhantes ao perfil de utilizador pretendido (seleção feita com base no organograma da empresa). De seguida foi-lhes solicitado que preenchessem um questionário de recrutamento que se encontra ilustrado nos anexos em A.4.

Através da análise das respostas a esse questionário, verificou-se que seis elementos correspondiam ao perfil pretendido, e foram então escolhidos para efetuar a avaliação.

Os principais fatores determinantes na seleção destes elementos foi a experiência de programação com a linguagem C, conhecimento sobre o domínio e a participação anterior em projetos na empresa relacionados com o standard ARINC 653. Este último fator serviu de confirmação que os utilizadores são conhecedores do domínio.

A complexidade dos projetos empresariais em que os elementos recrutados participaram, e uma curta entrevista informal efetuada após o preenchimento do questionário de recrutamento e da seleção dos seis elementos, serviu para selecionar e colocar os utilizadores em dois patamares distintos de experiência no domínio.

Os utilizadores participantes em projetos com complexidade entre 1 a 4 valores considerou-se que têm experiência média, e entre 5 e 6 valores considerou-se utilizadores com experiência elevada. Apesar de não ser habitual a consideração da escala desta forma, é necessário entender que devido à grande complexidade do domínio aeronáutico, um implementador que só por si participe num projeto oficial, independentemente da sua complexidade, já é um utilizador bastante conhecedor do domínio.

Um sétimo elemento, não pertencente ao grupo anterior de utilizadores, foi selecionado com intuito de esclarecer qual a dificuldade em utilizar a ferramenta para desenvolver uma aplicação *aviónica*, sendo um programador com média experiência, mas sem conhecimentos do domínio, nomeadamente, sobre o standard ARINC 653.

7.2.2 Preparação da Tarefa

Antes de se realizar a avaliação de usabilidade foi necessário preparar diversos materiais e organizar a mesma.

O questionário de recrutamento foi preparado logo de início, pode-se considerar que foi a etapa zero, pois o questionário foi importante para recrutar os utilizadores que efetuaram a avaliação. Todos os restantes materiais foram preparados nesta etapa. Alguns desses, já se encontravam disponíveis e foi só necessário dispô-los de forma apropriada. Outros, foram produzidos especificamente para a avaliação de acordo com o perfil de utilizador.

Principalmente produziram-se os slides que contêm a informação para realizar os exercícios propostos, bem como os enunciados dos exercícios e os questionários que foram respondidos pelos utilizadores no fim de cada sessão de avaliação.

Preparou-se o ambiente cuidadosamente para fornecer condições confortáveis, e equipamentos informáticos adaptados às necessidades dos utilizadores e dos exercícios que foram executados.

7.2.2.1 Slides

Os slides foram produzidos com intuito de guiar e auxiliar a apresentação da avaliação. Estes desempenharam um papel importante, pois serviram de item de consulta pelos utilizadores nos casos em que tiveram dúvidas sobre o domínio, sobre as DSLs ou sobre os exercícios.

Para cada uma das duas sessões de avaliação, desenvolveu-se um conjunto de slides, mas visto a segunda sessão ter acontecido imediatamente a seguir à primeira sessão, os slides desta só contêm o exercício que se realizou nessa segunda sessão. Os slides encontram-se ilustrados nos anexo em A.10, A.11, A.12 e A.13.

Na primeira sessão, os slides que foram disponibilizados aos utilizadores apresentam a seguinte estrutura:

- Introdução à arquitetura IMA, standard ARINC 653 e os seus componentes;
- Apresentação da interface gráfica de uma DSL, evidenciando-se os diversos elementos que estão presentes na linguagem;
- Sequência de imagens que mostram o desenvolvimento de uma aplicação *aviónica* com recurso a uma DSL;
- A descrição de um exercício para ser executado como treino;
- A descrição do exame da avaliação de usabilidade;
- Finalmente, um slide com os contactos do avaliador.

A introdução às tecnologias e componentes serviu exclusivamente para sanar algumas dúvidas que pudessem existir, pois os utilizadores lidam frequentemente só com alguns elementos do domínio e podem não se recordar de detalhes dos elementos que não utilizam no seu quotidiano de implementador.

Nos slides da apresentação da interface gráfica, explicitaram-se os diversos componentes que estavam visíveis na linguagem, bem como os seus efeitos quando pressionados ou arrastados. Os slides seguintes, com a sequência de imagens, permitiu demonstrar esses efeitos aos utilizadores e auxiliar o entendimento da semântica atribuída a cada elemento da linguagem, a forma de utilizá-los e como consequência, os resultados obtidos depois da sua utilização.

Foi disponibilizado um slide com um exercício que foi executado como treino e permitiu que os utilizadores experimentassem e entendessem mais aprofundadamente o funcionamento das DSLs. Posteriormente, apresentou-se um slide com o exame. Trata-se de um problema diferente do que foi apresentado como exercício de treino.

Finalmente, um slide onde se facultaram os contactos do avaliador, pois os utilizadores poderiam desejar expressar uma opinião à posteriori.

7.2.2.2 Exercícios

Foram produzidos três exercícios, o primeiro foi utilizado como treino, os outros dois como exame para a primeira e segunda DSL respetivamente. Em todos eles existem três sub exercícios que sobem consideravelmente de dificuldade, mas no final termina-se com uma aplicação *aviónica* complexa que incluiu lidar com o arranque de dois processos na partição, comunicação entre partições e comunicação entre processos da mesma partição.

No global, o exercício de treino é ligeiramente mais simples, por outro lado, os exercícios de exame de avaliação são idênticos mas utilizam portos de comunicação diferentes, pois neste caso o que se pretende avaliar é se as duas DSLs têm idêntica usabilidade ou não.

O exercício de treino e os exercícios de exame encontram-se disponíveis nos slides 16 e 17 do anexo A.12 e no slide 2 do anexo A.13.

Na figura 7.2 encontra-se ilustrada a solução para o exercício de treino. Como se pode observar, arrancar o processo *p1* e *p2*, bem como colocar a partição em modo *NORMAL* é a solução para o primeiro sub exercício.

O sub exercício seguinte é resolvido com os elementos que se encontram no processo *p1*. Obtém-se o id da *Queuing Port* para posteriormente indicar qual o porto onde se aguarda receber a mensagem. Seguidamente fica-se em ciclo a ler a informação da *Queuing Port* e a escrever a mesma informação no *Blackboard*.

De forma idêntica, o terceiro sub exercício é resolvido com os elementos que se encontram no processo *p2*. Obtém-se o id do *Blackboard* para posteriormente indicar qual o *Blackboard* onde se lê a mensagem. Seguidamente fica-se em ciclo a ler a informação do *Blackboard* e a escrever a mesma informação numa variável local.

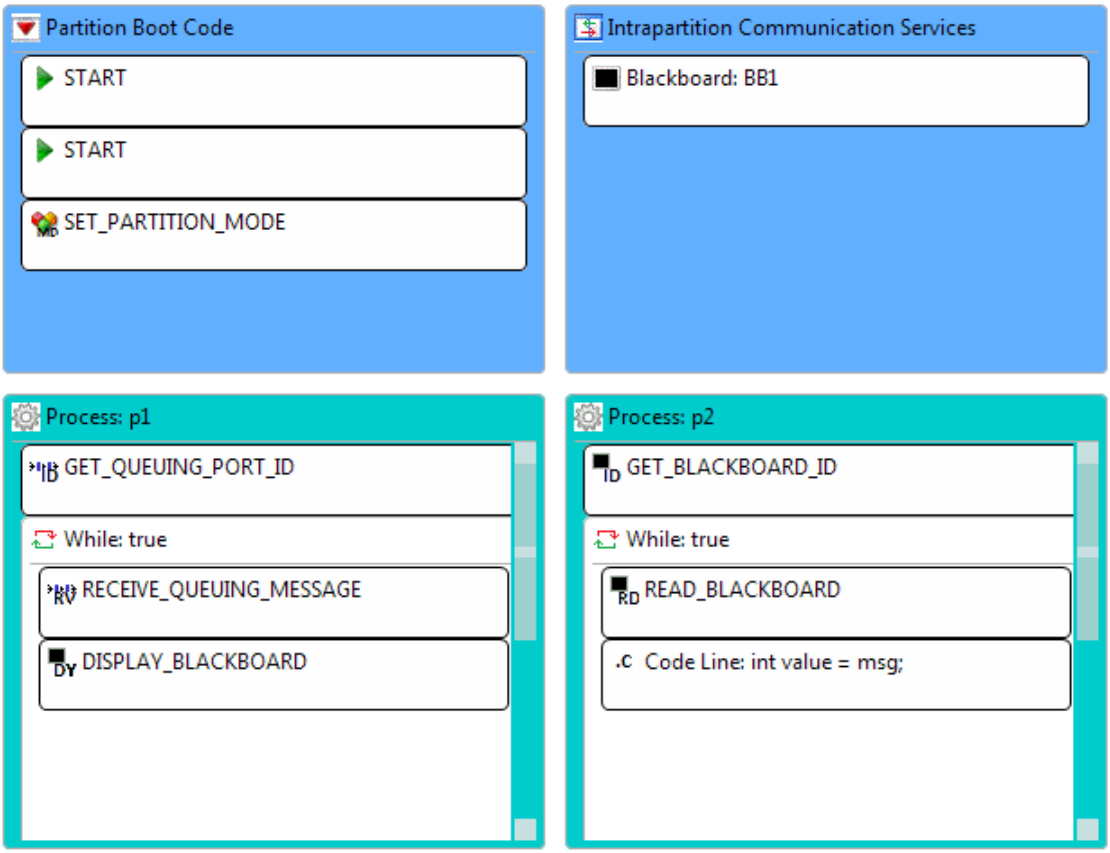


Figura 7.2: Solução do exercício de treino

7.2.2.3 Questionários

Solicitou-se que os utilizadores respondessem a três questionários. Um foi usado com intuito de recrutar os utilizadores mais adequados e já foi sucintamente explicado como a informação daí proveniente foi usada. Encontra-se na secção 7.2.1.

Os restantes dois questionários utilizaram-se principalmente para se entender a satisfação dos utilizadores com as DSLs testadas. Destes dois, o primeiro questionário está disponível nos anexos em A.5, A.6 e A.7. O segundo questionário está disponível nos anexos em A.8 e A.9.

No primeiro questionário (referente à primeira sessão) pretendeu-se confrontar a DSL com a linguagem C. A informação que se pretendeu obter individualmente em cada uma das questões foi a seguinte:

- **1ª Questão:** Verificar se o utilizador esteve descontente com a avaliação. A sua resistência ao processo, poderia influenciar os resultados obtidos;
- **2ª e 3ª Questão:** Comparar a confiança do utilizador no uso da DSL para auxiliar o desenvolvimento de uma aplicação, com a linguagem C normalmente usada. Entender se existiu alguma resistência à nova ferramenta;
- **4ª Questão:** Verificar indiretamente se o utilizador ficou satisfeito com a DSL e com os resultados por ela produzidos. Entender se a DSL é útil e facilita o desenvolvimento. Adicionalmente ajuda a compreender e/ou confirmar as respostas à 2ª e 3ª questão;
- **5ª Questão:** Reforça o entendimento das três questões anteriores. Permite averiguar se existe alguma resistência ao uso da DSL, bem como se os especialistas do domínio apreciaram a solução;
- **6ª Questão:** Reforça o entendimento das quatro questões anteriores. Permite averiguar se tecnicamente falta algum detalhe na DSL;
- **7ª Questão, Todas as alíneas:** Reforça o entendimento das questões anteriores. Permite confrontar a preferência dos utilizados face às duas abordagens seguidas;
- **8ª Questão:** Verificar se existe algum ícone que não forneça o entendimento claro sobre o seu significado;
- **9ª Questão:** Entender o nível de satisfação relativo à facilidade global de usar a DSL;
- **10ª Questão:** Entender o nível de satisfação relativo à confiança global de obter um bom resultado com o uso da DSL;
- **11ª Questão:** Verificar se a DSL exprime corretamente os conceitos do domínio. Entender se satisfaz as necessidades do utilizador. Confirmar se a expressividade das linguagens afetou a resposta às questões anteriores;

- **12ª e 13ª Questão:** Perguntas de resposta aberta com intuito de obter comentários gerais sobre a DSL.

No segundo questionário (referente à segunda sessão) pretendeu-se confrontar a usabilidade da DSL examinada na primeira sessão com a DSL da segunda sessão. A informação que se pretendeu obter individualmente em cada uma das questões foi a seguinte:

- **1ª Questão:** Verificar se o utilizador esteve descontente com a avaliação. A sua resistência ao processo, poderia influenciar os resultados obtidos;
- **2ª Questão:** Comparar a confiança do utilizador no uso da DSL e nos resultados por esta produzidos. Entender se existiu alguma resistência à nova ferramenta;
- **3ª Questão:** Entender o nível de satisfação relativo à facilidade global de usar a DSL;
- **4ª Questão:** Entender o nível de satisfação relativo à confiança global de obter um bom resultado com o uso da DSL;
- **5ª Questão:** Verificar se a DSL exprime corretamente os conceitos do domínio. Entender se satisfaz as necessidades do utilizador. Confirmar se a expressividade das linguagens afetou a resposta às questões anteriores;
- **6ª Questão:** Comparar a usabilidade das DSLs avaliadas na primeira e segunda sessão.

Como se pode verificar nos questionários em anexo, as perguntas que se efetuaram são objetivas, no entanto foram concretizadas de forma a não influenciar nenhum tipo de resposta pelo utilizador. Com a objetividade pretendeu-se facilitar o entendimento das questões e evitar aborrecer o especialista do domínio.

As perguntas são preferencialmente de escolha múltipla e com os valores bem definidos. Pretendeu-se incentivar que o utilizador respondesse de acordo com a sua real opinião e não aleatoriamente.

A escala usada para as questões de escolha múltipla foi de 1 a 6, exigiu-se que o utilizador refletisse suficientemente sobre a sua resposta. Deste modo não foi possível responder só o valor intermédio sem refletir sobre a questão (por exemplo numa escala de 1 a 5 poderia responder-se só 3 a todas as questões).

7.2.2.4 Ambiente e Equipamento de Trabalho

O ambiente onde se realizou a avaliação é importante, pois deve-se fornecer um ambiente confortável para o utilizador. Desta forma evita-se que o mesmo se sinta incentivado a realizar a avaliação de forma rápida e desleixada com intuito de terminar o processo o mais breve possível.

Noutro ponto de vista, um ambiente ou equipamentos de trabalho inadequados podem prejudicar a execução da avaliação ou os resultados obtidos [2].

Para evitar constrangimentos e perturbações, reservou-se uma sala espaçosa (excedendo com alguma margem o espaço necessário) numa zona sossegada do edifício da empresa, com uma temperatura aproximada de 21 graus e luz ambiente providenciada por duas janelas, sem a mesma refletir diretamente nos monitores. Em complemento, ligou-se a luz artificial do escritório para manter uma luminosidade mínima.

Cada utilizador efetuou a avaliação numa secretária individual sentado numa cadeira confortável. Distribuiu-se papel para rascunho, bem como caneta e lápis. Para cada utilizador estava disponível um computador com o *software* descrito na tabela 7.1 a executar no *hardware* descrito na tabela 7.2.

Ambiente de Operação do Sistema	**Plataformas de Trabalho**
Sistema Operativo	Microsoft Windows 7 Profissional
Software	JAVA v1.7.0 DSL Configurável Notepad++ CamStudio

Tabela 7.1: Descrição do *software* utilizado na avaliação de usabilidade

Tipo de dispositivo	**Componente**	**Informação detalhada**
Torre de Secretária	Frequência de Relógio RAM Espaço no Disco Rígido Conexão de Rede Velocidade da Rede Tipo de Portas USB	3,1 GHz 4 GB 250 GB Cabo 1 Gbps v2.0
Monitor	Tamanho Resolução Cores	19 Polegadas 1280x1024 16 Milhões
Rato	Conexão Configuração	Cabo 2 Botões com Roldana
Teclado	Conexão Padrão Região	Cabo QWERTY Português

Tabela 7.2: Descrição do *hardware* utilizado na avaliação de usabilidade

Para além da DSL configurável, instalou-se o Notepad++[1] onde os utilizadores desenvolveram de raiz as aplicações *aviónicas* ou completaram o esqueleto das mesmas utilizando-se a linguagem C. Com intuito de se gravar o ecrã do utilizador para facilitar a contagem de enganos e erros dos especialistas do domínio, usou-se o CamStudio[2].

[1] http://notepad-plus-plus.org/
[2] http://camstudio.org/

7.2.3 Sessão Piloto

Para se evitarem problemas de última hora, que poderiam ter acontecido durante a execução do exame de avaliação de usabilidade, foi importante praticar-se pelo menos uma vez antes desse exame ocorrer.

Primeiro, o avaliador colocou-se no lugar do avaliado e executou a sessão de treino e o exame. Foram encontrados erros, mas não foi suficiente, porque o avaliador encontrava-se viciado na própria avaliação que preparou.

Posteriormente recrutou-se um utilizador distinto dos sete utilizadores que colaboraram na avaliação de usabilidade. Este utilizador correspondeu ao perfil de especialista do domínio e efetuou a secção de treino e exame. Com esta sessão piloto foi possível encontrar erros na documentação auxiliar fornecida, bem como corrigir a posição das secretárias, pois o reflexo da luz exterior proveniente das janelas incidia ligeiramente nos monitores. Adicionalmente, permitiu obter uma estimativa mais precisa em relação ao tempo necessário para executar a avaliação.

7.2.4 Sessão de Treino

A sessão de treino foi onde os utilizadores tiveram o primeiro contacto com a DSL e aconteceu imediatamente antes do exame, mas depois de terem sido apresentados os slides da primeira sessão de avaliação.

Nesta sessão os utilizadores tiveram a oportunidade de experimentar a linguagem e posteriormente executar um exercício disponível no slide 16 do anexo A.12.

Apesar do exercício de treino não ter sido considerado como avaliação, foi pedido aos utilizadores que também aceitassem gravar o ecrã. Desta forma, foi possível retirar e considerar os tempos que levaram a realizar os sub exercícios. Consequentemente, foi possível analisar os tempos de aprendizagem e adaptação, bem como verificar a curva de aprendizagem das linguagens produzidas.

7.2.5 Exame

A avaliação de usabilidade foi efetuada com recurso a duas sessões por utilizador. Na primeira sessão o utilizador desenvolveu a aplicação com recurso à DSL para gerar o esqueleto de código da aplicação e posteriormente completou-o com recurso à linguagem C. Nessa sessão o utilizador, também desenvolveu a mesma aplicação *aviónica* utilizando só a linguagem C.

Na segunda sessão, o utilizador testou unicamente a interface gráfica da segunda DSL, ou seja, utilizou a linguagem para representar um problema idêntico ao testado na primeira sessão mas desta vez, de acordo com a configuração da segunda linguagem.

Os utilizadores foram divididos em grupos de dois elementos excetuando-se o utilizador sem experiência que efetuou a avaliação sozinho. Este não apresentava o perfil desejado, foi recrutado com intuito de esclarecer qual a dificuldade que um utilizador

deste tipo teria em usar a DSL. Nesse sentido, teve-se a precaução em explicar aprofundadamente os conceitos do domínio, e executou-se a avaliação, primeiro solicitando que o utilizador usasse a DSL para desenvolver o esqueleto da aplicação *aviónica*, completando posteriormente o mesmo com código C. Em segundo solicitou-se que este desenvolvesse o mesmo *software* usando só a linguagem C. Esta sequência de avaliação facilita o desenvolvimento da aplicação quando é usada só a linguagem C, porque já existe como referência a aplicação desenvolvida com auxílio da DSL.

Os restantes seis utilizadores são os elementos relevantes para esta avaliação, no entanto os utilizadores experientes são o foco central, pois são normalmente os responsáveis por desenvolverem as aplicações de elevada criticidade. Devido a este facto, os quatro elementos com elevada experiência e divididos em grupos de dois elementos, executaram a primeira sessão de forma diferente.

Um grupo de dois utilizadores experientes utilizou primeiro a DSL e posteriormente completou o esqueleto da aplicação com código C. Seguidamente desenvolveram a mesma aplicação utilizando só a linguagem C.

Com o segundo grupo, inverteu-se a ordem, primeiro desenvolveram o *software* utilizando só a linguagem C. Posteriormente usaram a DSL e completaram o esqueleto da aplicação com código C.

Esta inversão na sequência permitiu analisar e verificar se os resultados foram influenciados pela ordem que a avaliação foi executada, e assim, desta forma atenuar os seus efeitos.

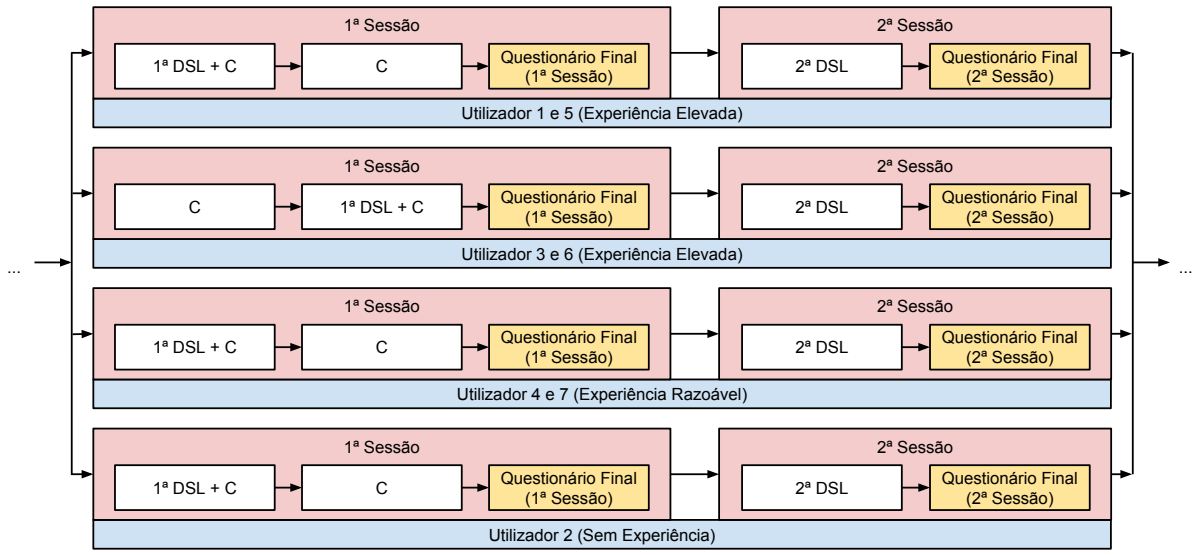


Figura 7.3: Sequências de Sessões de Avaliação da Usabilidade

Na figura 7.3 encontra-se representada a sequência de sessões de avaliação, bem como os utilizadores que a realizaram. Por motivos de privacidade e uma vez que o principal objetivo desta avaliação foi exclusivamente avaliar as linguagens, o nome dos utilizadores foi ocultado. Os novos nomes simbólicos mantêm a correta coerência com os nomes

apresentados na secção 7.3, bem como com os dados obtidos com a avaliação de usabilidade.

Foram concretizadas quatro avaliações de forma isolada. Cada avaliação tem duas sessões e realizou-se sempre com o mesmo avaliador. Durante a avaliação não se esclareceram dúvidas, desta forma pretendeu-se não influenciar os resultados obtidos.

Finalmente foi crucial perceber-se a satisfação dos utilizadores, para isso solicitou-se que respondessem a questionários. Nesse sentido a informação que se obteve facilitou uma melhor compreensão das métricas obtidas durante o processo de avaliação. Noutra perspetiva, recebeu-se comentários construtivos que permitem otimizar o *software* para ir de encontro às necessidades dos utilizadores.

Este questionário foi entregue em papel e respondido pelos utilizadores no final de cada sessão de avaliação. O questionário que se encontra disponível nos anexos em A.5, A.6 e A.7 foi respondido no final da primeira sessão. De forma idêntica, o questionário que se encontra disponível nos anexos em A.8 e A.9 foi respondido no final da segunda sessão.

7.3 Análise dos Resultados

Nesta secção apresentam-se as informações recolhidas com a realização da avaliação de usabilidade. Não menos importante, apresenta-se também uma análise dos dados obtidos. De acordo com o processo ilustrado na figura 7.1, esta secção corresponde à etapa número seis.

Grupo	Utilizadores	Experiência
1	1 e 5	Elevada
2	3 e 6	Elevada
3	4 e 7	Razoável
4	2	Nenhuma

Tabela 7.3: Identificação dos grupos

Na tabela 7.3 clarificam-se os grupos de utilizadores que estão mencionados na figura 7.3. Doravante a exposição dos dados é efetuada considerando-se os grupos em detrimento dos utilizadores individuais.

7.3.1 Tempo de Aprendizagem

Através da curva de aprendizagem mede-se a facilidade dos utilizadores aprenderem uma linguagem. Esta foi calculada com a informação obtida através da medição do tempo despendido pelos utilizadores a realizar os sub exercícios da sessão de treino (ver na secção 7.2.4).

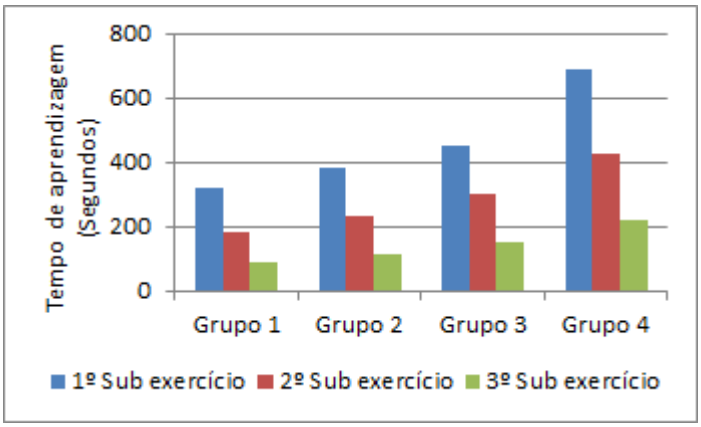


Figura 7.4: Tempo de aprendizagem

Através da análise do gráfico apresentado na figura 7.4, verifica-se que apesar da complexidade dos sub exercícios aumentar desde o primeiro até ao terceiro exercício, os grupos reduzem significativamente o tempo necessário para os concretizar. Além disso, verifica-se que nenhum dos grupos desistiu de efetuar os exercícios. Este facto é ainda mais importante, considerando que o grupo 4 não é composto por um especialista, mas sim, por um utilizador sem experiência no domínio.

Quando se observam os valores do grupo 4 referente ao terceiro exercício, verifica-se que este apresenta valores próximos dos restantes grupos compostos por especialistas do domínio, no entanto, é evidente que no primeiro sub exercício o tempo consumido foi superior aos restantes grupos devido à ausência de experiência.

De acordo com os valores obtidos, considera-se que a DSL avaliada apresenta características que facilitam a aprendizagem, mesmo que o utilizador não seja especialista do domínio.

7.3.2 Erros Efetuados

Através da contabilização do número de erros efetuados durante o desenvolvimento, é possível avaliar a eficácia de uma linguagem. Nesta secção apresenta-se o estudo realizado nesse âmbito.

No decorrer da etapa de análise dos resultados, verificou-se que os utilizadores efetuaram um elevado número de erros quando utilizavam a linguagem C para completar o esqueleto de código proveniente da DSL, mas também quando desenvolviam a aplicação de raiz.

Depois de um exame profundo a este caso, verificou-se que os valores obtidos foram influenciados por uma decisão incorreta no planeamento da avaliação. Concretamente, forneceu-se aos utilizadores uma ferramenta inadequada. Quando os especialistas tinham de desenvolver código C, executavam essa tarefa no Notepad++, trata-se de uma aplicação de texto comum, sem validação como tem a DSL ou um IDE.

Para contornar esta situação decidiu-se fazer um levantamento completo dos erros

realizados pelos utilizadores. Finalmente, não se contabilizaram os erros produzidos pela ausência de um IDE.

Na listagem seguinte, descrevem-se os erros que não foram contabilizados:

- Invocação de funções com argumentos em excesso;
- Invocação de funções com argumentos em falta;
- Declaração de variáveis que não são usadas;
- Declaração de variáveis com o tipo incorreto;
- Uso de variáveis não inicializadas;
- Erros no nome das instruções (mas foi invocada a instrução correta).

Considerando que o objetivo da DSL é produzir um esqueleto de código C que posteriormente é completado, julgou-se mais relevante considerar os erros e os enganos do processo completo, ou seja, os dados que serão apresentados consideram o desenvolvimento completo da aplicação *aviónica*. Por exemplo, no caso da DSL apresenta-se o somatório dos erros de utilização da mesma com os erros ocorridos quando o esqueleto foi completado com código C.

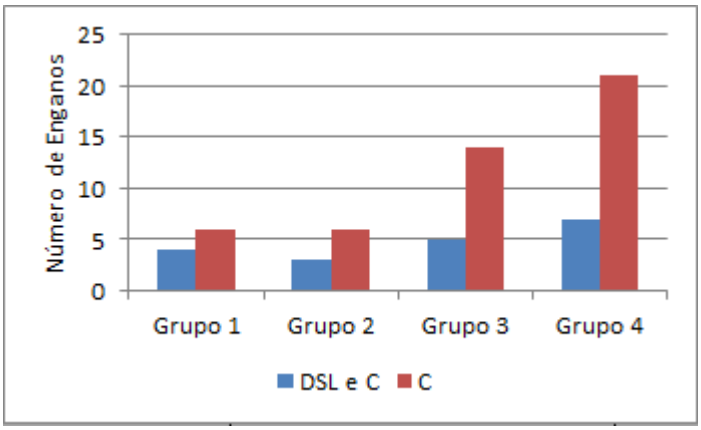


Figura 7.5: Número de enganos

No gráfico da figura 7.5 estão representados o número de enganos. Consideram-se enganos os erros que o próprio utilizador detetou e corrigiu num curto período de tempo.

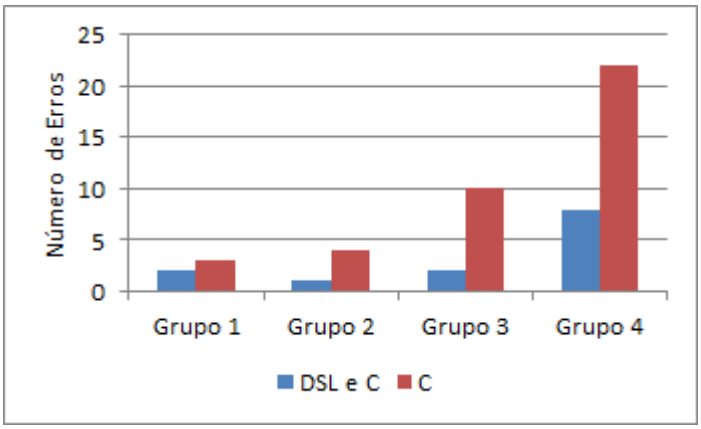


Figura 7.6: Número de erros

No gráfico da figura 7.6 estão representados o número de erros. Consideram-se erros, os erros que o utilizador efetuou e não detetou nem corrigiu durante o período de tempo da sessão de avaliação.

Através da análise dos gráficos 7.5 e 7.6, verifica-se que utilizando a DSL reduz-se significativamente o número de enganos e erros em todos os grupos de utilizadores, mostrando a eficácia da linguagem.

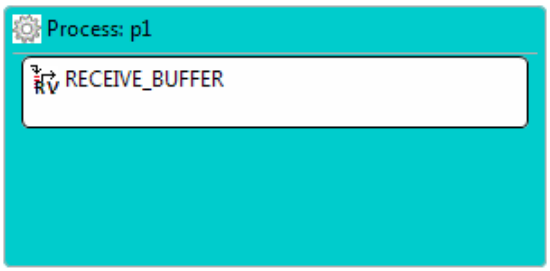


Figura 7.7: Exemplo de erro cometido pelo utilizador

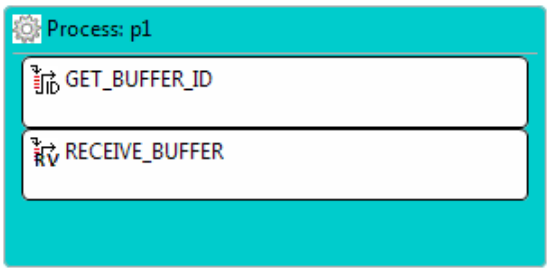


Figura 7.8: Correção do exemplo de erro cometido pelo utilizador

O erro mais frequente na utilização da DSL encontra-se ilustrado na figura 7.7. Apresenta-se a correta utilização da mesma na figura 7.8.

O utilizador, quando pretende usar uma *Sampling Port*, *Queuing Port*, *Blackboard*, *Buffer*, *Event* ou *Semaphore* num processo, necessita de obter o identificador pelo menos uma vez. A linguagem parece sugerir ao especialista do domínio que efetua essa operação de

forma automática.

Refletiu-se sobre o erro anterior, e uma possível solução passa por adicionar automaticamente a operação ARINC 653 para obter o identificador no interior do processo, sempre que se utiliza outra operação ARINC 653 relacionada.

7.3.3 Tempo Consumido

Através da contabilização do tempo consumido no desenvolvimento, é possível avaliar a eficiência de uma linguagem. Nesta secção apresenta-se o estudo realizado nesse contexto.

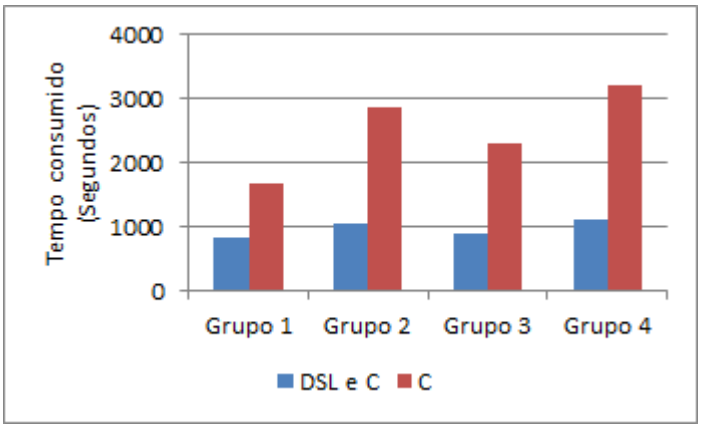


Figura 7.9: Tempo consumido

Com a análise do gráfico presente na figura 7.9, verifica-se que a DSL torna a produção de *software aviónico* mais eficiente, nomeadamente calculou-se o valor médio de ganho da eficiência e obteve-se o valor de 59,91%.

No desenvolvimento utilizando só a linguagem C, o grupo 2 constituído por especialistas experientes, consumiu mais tempo que o grupo 3 constituído por especialistas razoáveis. Este facto, verificou-se, porque um dos utilizadores foi mais perfecionista no código que produziu, por exemplo indentou e comentou o código corretamente.

7.3.4 Respostas dos Questionários

A satisfação dos utilizadores é avaliada através das respostas aos questionários que foram preenchidos no final dos testes de usabilidade. Nesta secção, consideram-se os dados provenientes da segunda sessão onde foi testada a segunda DSL.

Através dos questionários, obteve-se diversas informações relevantes, mas principalmente com as respostas às questões 9, 10 e 11 do questionário da primeira sessão, consegue-se percecionar a avaliação global às DSLs. Estas questões estão cotadas de 1 a 6, e a pontuação obtida em cada questão está ilustrada no gráfico da figura 7.10. O significado destas questões encontram-se explicitadas na secção 7.2.2.3.

De uma forma global, é possível percecionar que os utilizadores, acharam a DSL fácil, confiável e expressiva.

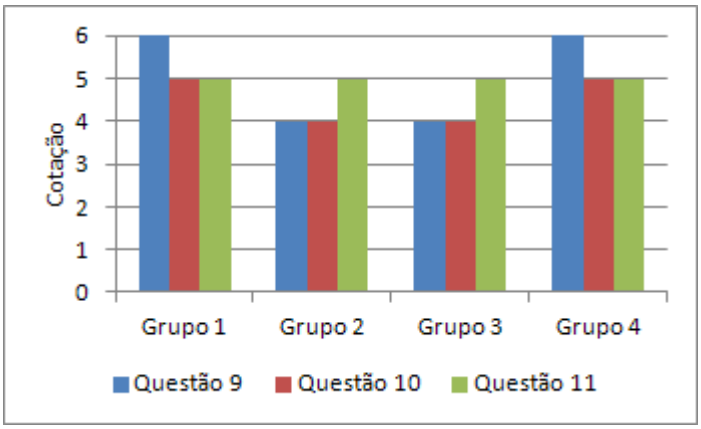


Figura 7.10: Cotação das questões 9, 10 e 11 do 1º Questionário

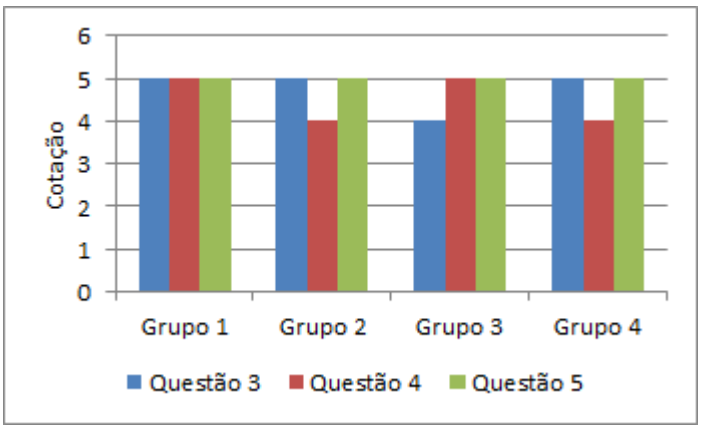


Figura 7.11: Cotação das questões 3, 4 e 5 do 2º Questionário

As questões 9, 10 e 11 do questionário da primeira sessão são equivalentes às questões 3, 4 e 5 do questionário da segunda sessão. Desta forma é possível comparar os resultados obtidos, que se encontram ilustrados no gráfico da figura 7.11. Verifica-se que os valores das respostas são idênticos, logo o nível de satisfação com as linguagens produzidas é estável, e aplica-se a todas as DSL da mesma família.

As respostas à questão 6 do questionário da segunda sessão, confirma essa consideração, pois foi perguntado se a segunda DSL apresenta menor, igual ou maior nível de usabilidade, e todos os utilizadores reponderam que apresenta o mesmo nível de usabilidade.

Em relação à questão 8 do questionário da primeira sessão, onde se pretende desvendar se existe algum ícone que seja difícil de compreender o seu significado, houve poucas respostas negativas, provavelmente porque os utilizadores guiam-se pelo nome das operações ARINC 653.

Na questão 12 e 13 desse mesmo questionário, os especialistas do domínio tiveram oportunidade de escrever comentários. Principalmente, os utilizadores consideram importante alterar o modelo de interação para *drag and drop*, validação constante efetuada a qualquer pequena alteração, preenchimento automático com valores por defeito em

algumas propriedades e finalmente a existência de uma lista global de avisos e erros.

7.4 Ameaças à Validade

Nesta secção demonstram-se as limitações, possíveis influências e ameaças que podem ter comprometido a avaliação de usabilidade realizada. No total foram efetuadas quatro avaliações com duas sessões de avaliação cada uma. Durante as mesmas verificaram-se fatores externos imprevisíveis que são difíceis e/ou impossíveis de controlar e que aqui serão relatados. Adicionalmente, descrevem-se decisões que foram determinadas para a experimentação, mas que não foram as mais indicadas.

As avaliações foram realizadas com sete elementos, sendo que só seis são verdadeiramente importantes para este estudo. O reduzido número de utilizadores apresenta-se como um fator negativo e foi considerado mesmo antes de ser realizada esta avaliação, no entanto devido ao domínio ser bastante restrito, foi impossível contornar esta limitação.

Por outro lado os resultados expressivos dos questionários, demonstraram que os utilizadores ficaram satisfeitos com as vantagens que as DSLs permitem obter. Este facto é surpreendente, pois são utilizadores habituados a lidar com interfaces do tipo linha de comandos, o que permite supor que o efeito de uma amostra reduzida nos resultados da avaliação, pode ser de pequena dimensão.

Outro fator relevante prende-se com a proximidade de todos os elementos recrutados, ou seja, todos os utilizadores trabalham na mesma empresa. Desta forma, depois dos elementos terminarem a sua avaliação e voltarem para o exterior da sala, conversavam com utilizadores que iriam mais tarde ser submetidos à mesma avaliação. Este fator não foi previsto e pode implicitamente ter influenciado a avaliação, no entanto tal não se observou nos resultados obtidos.

Desde a etapa de planeamento que se decidiu que os utilizadores para escreverem o código C usariam o Notepad++, esta aplicação é um editor de texto comum. Verificou-se que não foi a opção mais correta, pois as DSLs tinham validação e o editor de texto não. Assim os resultados das DSLs sairiam beneficiados, porque escrever o código C sem qualquer ajuda adicional provoca mais erros. Para minorar esta situação, não se contabilizaram os erros que poderiam ter sido evitados caso tivesse sido utilizado um IDE.

Como referido no início deste capítulo não foi utilizado o gerador de DSLs, mas sim a DSL configurável. Se o gerador fosse utilizado para gerar as DSLs, este iria consumir mais tempo numa fase inicial da avaliação, esta situação poderia gerar alguma impaciência nos utilizadores mas seria reduzida, porque só iriam gerar duas linguagens (uma por cada sessão) e o tempo necessário para gerar uma linguagem ronda os três minutos.

Finalmente, não foi possível testar todos os elementos gráficos das linguagens, pois resultaria em exercícios demasiado longos e complicados. Dessa forma, seria necessário despender muito tempo para realizar cada sessão, o que poderia cansar e desmotivar os utilizadores.

7.5 Considerações

A avaliação de usabilidade apresentou um resultado global positivo. Por um lado todos os utilizadores apreciaram a nova ferramenta, principalmente porque lhes facilitou a geração de código bastante repetitivo e que desenvolvem com frequência manualmente. Por outro porque gerou o código C já estruturado e só tiveram de preencher alguns espaços com código manualmente escrito. Desta forma tiveram de raciocinar menos sobre o *software* que se encontravam a desenvolver.

Através da curva de aprendizagem verificou-se que quando se conhece minimamente os conceitos do domínio, então a linguagem é fácil de aprender, mesmo para utilizadores inexperientes e pouco conhecedores do standard ARINC 653.

Na contabilização de erros também se verificou um tendência positiva, ou seja, quando se utiliza a DSL, por regra produz-se menos erros.

De igual forma, os tempos de desenvolvimento são mais reduzidos quando se utiliza a DSL, com uma diferença média a rondar os 60%, no entanto esperava-se uma diferença maior. Este facto pode se ter verificado porque os exercícios e exames resolvidos são de pequena dimensão comparado com um *software* real. Além disso, a DSL foi avaliada conjuntamente com a linguagem C, atenuando o efeito benéfico da mesma.

8

Conclusões

8.1 Sumário da Dissertação

A presente dissertação iniciou-se com um desafio ambicioso. Pretendeu-se construir uma ferramenta para a aeronáutica que limitasse a possibilidade dos especialistas do domínio cometerem erros no desenvolvimento de aplicações *aviónicas*. Simultaneamente requeria-se que esta ferramenta permitisse diminuir o tempo de implementação, bem como o número de erros cometidos.

Uma DSL apresentou-se como solução provável, no entanto não é viável produzir-se uma linguagem nova para cada configuração diferente de uma partição que execute num módulo *aviónico*.

Aparentemente num beco sem saída, inspirou-se nos conceitos e definição de SPL para se definir o conceito de família de DSLs. *Trata-se de um conjunto de linguagens para um domínio específico, que apresentam um conjunto comum de conceitos chave, mas que adaptam alguns desses conceitos para cumprir a variabilidade dos requisitos*.

Investigou-se qual a melhor forma de produzir automaticamente uma família de linguagens. Decidiu-se que a variabilidade positiva se apresenta como a mais indicada para o complexo domínio IMA, pois permite construir o metamodelo da linguagem através da fusão do metamodelo dos conceitos comuns do domínio com o metamodelo dos conceitos de configuração. Este último é produzido de acordo com a configuração da partição *aviónica*.

Com base nesta conceptualização, construiu-se um protótipo completo e funcional, chamou-se gerador de DSLs e foi batizado de *IMA Studio*. Este gerador produz o metamodelo da linguagem e os restantes artefactos necessários.

Durante esta dissertação verificou-se que este conceito aplica-se a outros domínios, por exemplo na secção 5.3 foi demonstrado que pode ser aplicado num robô. No geral, provou-se o conceito de fábrica de linguagens, aplicou-se o mesmo a um domínio complexo, e demonstrou-se que a ideia de fábrica automática de DSLs pode ser aplicada

também a domínios simples.

As linguagens produzidas pelo gerador de DSLs, apresentam uma representação forte dos conceitos do standard ARINC 653. Este facto deve-se às características dos utilizadores alvo, pois são especialistas habituados a desenvolver aplicações usando a linguagem C. Estes foram consultados e decidiu-se que é a melhor forma de as DSLs representarem as abstrações do domínio e serem usáveis em ambiente real.

Efetuou-se uma avaliação de usabilidade e obteve-se três resultados importantes. Primeiro verificou-se que as duas DSLs avaliadas apresentam um nível de usabilidade idêntico, desta forma prova-se que o gerador de DSLs produz linguagens coerentes e usáveis. Segundo verificou-se que as linguagens representam corretamente as abstrações do domínio. Finalmente, que usando as DSLs produz-se menos erros nas aplicações desenvolvidas, bem como diminui-se o tempo de implementação.

8.2 Contribuições

Com a realização da dissertação alcançaram-se diversas contribuições. No âmbito científico efetuou-se a definição precisa do conceito de famílias de DSLs e categorizaram-se as diversas abordagens para o seu desenvolvimento. Não menos importante, desenvolveu-se um protótipo que demonstra a concretização deste mesmo conceito.

Ao nível empresarial implementou-se um protótipo que nasceu do gerador de DSLs, e foi incluído no *Integrated Modular Avionics Development Environment* (IMADE) [15, 48], bem como, disseminou-se a utilização de práticas de modelação na empresa.

8.3 Trabalho Futuro

Acredita-se que a presente dissertação contribuiu com uma importante abordagem que faz uso de um conceito inovador. Neste âmbito deparou-se com algumas tarefas que devem ser realizadas como continuação do trabalho já concretizado.

Considerando o protótipo gerador de DSLs, este pode ser modificado manualmente para comportar evoluções do domínio, no entanto uma nova abordagem relevante, seria produzir este gerador através de transformações, ou seja, um gerador de geradores de DSLs. Nesse caso, seria necessário recorrer a *Higher-Order Transformations* (HOT).

Esta abordagem não seria útil só para evolução de um gerador cingido a um determinado domínio, mas também pode ser visualizado como um gerador de geradores de DSLs para diferentes domínios, ou seja, uma fábrica de geradores capazes de cada um individualmente se aplicar a domínios diferentes.

Nesta dissertação sentiu-se diversas dificuldades e desafios relacionados com o suporte fornecido pelas *Languages Metamodeling Workbench* (LMW) para implementar o protótipo. Neste sentido é necessário complementar as LMWs para suportar uma melhor orquestração de tarefas, bem como fornecer um sistema integrado que permita desenvolver funcionalidade de acesso à informação exterior. Adicionalmente, é necessário otimizar o

desempenho dos mecanismos internos responsáveis por executar as transformações. Este último fator, foi a motivação para o desenvolvimento do protótipo empresarial explicado na secção 6.2.

Finalmente, as DSLs produzidas pelo protótipo podem ser melhoradas considerando as opiniões fornecidas pelos especialistas do domínio quando se realizou a avaliação de usabilidade.

Bibliografia

[1] Abdoulaye Gamatié, Christian Brunette, Romain Delamare, Thierry Gautier e Jean-Pierre Talpin. *A Modeling Paradigm for Integrated Modular Avionics Design*. Rel. téc. INRIA, 2005.

[2] Ankica Barišić, Pedro Monteiro, Vasco Amaral, Miguel Goulão e Miguel Monteiro. "Patterns for Evaluating Usability of Domain-Specific Languages". Em: *Proceedings of the Pattern Languages of Programs Conference*. PLoP 2012. ACM, 2012.

[3] Ankica Barišić, Vasco Amaral e Miguel Goulão. "Usability Evaluation of Domain-Specific Languages". Em: *Proceedings of the 8th International Conference on the Quality of Information and Communications Technology (QUATIC 2012)*. SEDES. IEEE Computer Society, 2012.

[4] Ankica Barišić, Vasco Amaral, Miguel Goulão e Bruno Barroca. "How to Reach a Usable DSL? Moving Toward a Systematic Evaluation". Em: *Proceedings of the 6th Workshop on Multi-paradigm Modeling - MODELS 2011*. Electronic Communications of the EASST. EASST, 2011.

[5] Ankica Barišić, Vasco Amaral, Miguel Goulão e Bruno Barroca. "Quality in Use of Domain-specific Languages: A Case Study". Em: *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*. PLATEAU '11. ACM, 2011.

[6] Anneke Kleppe, Jos Warmer e Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston, MA, USA: Addison-Wesley, 2003.

[7] *ARINC 429 Digital Information Transfer System (DITS), Part 1, Functional Description, Electrical Interfaces, Label Assignments and Word Formats*. Annapolis, Maryland, USA: Aeronautical Radio, Inc., 1977.

[8] *ARINC 650 Integrated Modular Avionics Packaging and Interfaces*. Annapolis, Maryland, USA: Aeronautical Radio, Inc., 1994.

[9] *ARINC 651 Design Guidance for Integrated Modular Avionics*. Annapolis, Maryland, USA: Aeronautical Radio, Inc., 1997.

[10] *ARINC 653 Avionics Application Software Standard Interface, Part 1, Required Services*. Annapolis, Maryland, USA: Aeronautical Radio, Inc., 2003.

[11] *ARINC 653 Avionics Application Software Standard Interface, Part 2, Extended Services*. Annapolis, Maryland, USA: Aeronautical Radio, Inc., 2003.

[12] *ARINC 664 Aircraft Data Network, Part 1, Systems Concepts and Overview*. Annapolis, Maryland, USA: Aeronautical Radio, Inc., 2006.

[13] Arun Sen. "The Role of Opportunism in the Software Design Reuse Process". Em: *IEEE Transactions on Software Engineering* 23.7 (1997).

[14] Bruno Barroca, Levi Lúcio, Didier Buchs, Vasco Amaral e Luís Pedro. "DSL Composition for model-based test generation". Em: *3rd International Workshop on Multi-Paradigm Modeling: Concepts and Tools*. Ed. por Tihamer Levendovszky, László Lengyel, Gabor Karsai e Cécile Hardebolle. Electronic Communications of the EASST 21. EASST, 2009.

[15] Bruno Tavares, João Cintra e Ricardo Alves. "IMADE: Integrated Modular Avionic Development Environment". Em: *Proceedings of the Digital Avionics Systems Conference, 2014. DASC '14. IEEE/AIAA 33rd*. Washington, DC, USA: IEEE Computer Society, 2014.

[16] Changyun Huang, Yasutaka Kamei, Kazuhiro Yamashita e Naoyasu Ubayashi. "Using Alloy to Support Feature-based DSL Construction for Mining Software Repositories". Em: *Proceedings of the 17th International Software Product Line Conference Co-located Workshops*. SPLC '13 Workshops. Tokyo, Japan: ACM, 2013.

[17] Cláudio Gomes. "A Framework for Efficient Model Transformations". Tese de mestrado. Faculdade de Ciências e Tecnologia - Universidade Nova de Lisboa, 2013.

[18] Cláudio Silva. "Integrated Modular Avionics for Space Applications: Input/Output Module". Tese de mestrado. Universidade Técnica de Lisboa - Instituto Superior Técnico, 2012.

[19] David Steinberg, Frank Budinsky, Marcelo Paternostro e Ed Merks. *EMF: Eclipse Modeling Framework*. 2nd. Addison-Wesley Professional, 2009.

[20] Dimitrios Kolovos, Richard Paige e Fiona Polack. "Merging Models with the Epsilon Merging Language (EML)". Em: *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems*. MODELS 2006. Berlin, Heidelberg: Springer-Verlag, 2006.

[21] *DO-178C Software Considerations in Airborne Systems and Equipment Certification*. Washington, DC, USA: RTCA, Inc., 2011.

[22] Hassan Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Boston, MA, USA: Addison-Wesley, 2004.

[23] *ISO 9241-11:1998 Ergonomic Requirements for Office Work with Visual Display Terminals (VDTs), Part 11, Guidance on Usability*. Geneve, Switzerland: International Organization for Standardization, 1998.

[24] Jean-Bernard. *A380 Integrated Modular Avionics: The History, Objectives and Challenges of the Deployment of IMA on A380*. Rel. téc. Airbus S.A.S., 2010.

[25] João Craveiro. "Integration of Generic Operating Systems in Partitioned Architectures". Tese de mestrado. Universidade De Lisboa - Faculdade de Ciências, 2009.

[26] Jules White, James Hill, Sumant Tambe, Aniruddha Gokhale, Douglas Schmidt e Jeff Gray. *Improving Domain-specific Language Reuse through Software Product-line Configuration Techniques*.

[27] Kevin Forsberg e Harold Mooz. "The Relationship of System Engineering to the Project Cycle". Em: *Proceedings of the First Annual Symposium of National Council on System Engineering*. Chattanooga, TN, USA, 1991.

[28] Klaus Pohl, Günter Böckle e Frank Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[29] Krzysztof Czarnecki e Chang Kim. "Cardinality-Based Feature Modeling and Constraints: A Progress Report". Em: (2005).

[30] Krzysztof Czarnecki e Simon Helsen. "Feature-based Survey of Model Transformation Approaches". Em: *IBM Syst. J.* 45.3 (2006).

[31] Kyo Kang, Sholom Cohen, James Hess, William Novak e A. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Rel. téc. CMU/SEI-90-TR-021. Software Engineering Institute, 1990.

[32] Luis Pedro. "A Systematic Language Engineering Approach for Prototyping Domain Specific Modelling Languages". Tese de doutoramento. Université de Genève - Faculté des Sciences, 2009.

[33] Luis Pedro, Didier Buchs e Vasco Amaral. "Foundations for a Domain Specific Modeling Language Prototyping Environment: A compositional approach". Em: *Proceedings of the 8th OOPSLA ACM-SIGPLAN Workshop on Domain-Specific Modeling (DSM)*. 2008.

[34] Marjan Mernik, Jan Heering e Anthony Sloane. "When and How to Develop Domain-Specific Languages". Em: *ACM Computing Surveys* 37.4 (2005).

[35] Markus Voelter. "A Family of Languages for Architecture Description". Em: *in 8th OOPSLA Workshop on Domain-Specific Modeling (DSM'08)*. 2008.

[36] Markus Voelter e Iris Groher. "Product Line Implementation Using Aspect-Oriented and Model-Driven Software Development". Em: *Proceedings of the 11th International Software Product Line Conference*. SPLC '07. Washington, DC, USA: IEEE Computer Society, 2007.

[37] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart Kats, Eelco Visser e Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[38] Martin Fowler. *Domain Specific Languages*. Addison-Wesley, 2010.

[39] Mikaël Barbero, Frédéric Jouault, Jeff Gray e Jean Bézivin. "A Practical Approach to Model Extension". Em: *Proceedings of the 3rd European Conference on Model Driven Architecture-foundations and Applications*. ECMDA-FA'07. Berlin, Heidelberg: Springer-Verlag, 2007.

[40] Pablo Sanchez, João Santos, Mauricio Alférez, Awais Rashid, Lidia Fuentes, Ana Moreira, João Araújo e Uirá Kulesza. "VML* – A Family of Languages for Variability Management in Software Product Lines". Em: *Proceedings of the International Conference on Software Language Engineering (SLE)*. ACM Press, 2009.

[41] Paul Clements e Linda Northrop. *Software Product Lines: Practices and Patterns*. Boston, MA, USA: Addison-Wesley, 2002.

[42] Paul Guernic, Thierry Gautier, Michel Borgne e Claude Maire. "Programming Real-time Applications with SIGNAL". Em: (1991).

[43] Pedro Leonardo. "Child Programming: An Adequate Domain Specific Language for Programming Specific Robots". Tese de mestrado. Faculdade de Ciências e Tecnologia - Universidade Nova de Lisboa, 2013.

[44] Peter Swithinbank, Mandy Chessell, Tracy Gardner, Catherine Griffin, Jessica Man, Helen Wylie e Larry Yusuf. *Patterns: Model-driven Development Using Ibm Rational Software Architect*. Riverton, NJ, USA: IBM Corp., 2005.

[45] Richard Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 2009.

[46] Shane Sendall e Wojtek Kozaczynski. "Model Transformation: The Heart and Soul of Model-Driven Software Development". Em: (2003).

[47] Thomas Kühne. "What is a Model?" Em: *Language Engineering for Model-Driven Software Development*. Ed. por J. Bezivin e R. Heckel. Dagstuhl Seminar Proceedings 04101. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, 2005.

[48] Tobias Schoofs, Sérgio Santos, Cássia Tatibana e José Anjos. "An Integrated Modular Avionics Development Environment". Em: *Proceedings of the Digital Avionics Systems Conference, 2009. DASC '09. IEEE/AIAA 28th*. Washington, DC, USA: IEEE Computer Society, 2009.

[49] Walter Cazzola e Davide Poletti. "DSL Evolution Through Composition". Em: *Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution*. RAM-SE '10. Maribor, Slovenia: ACM, 2010.

Anexos

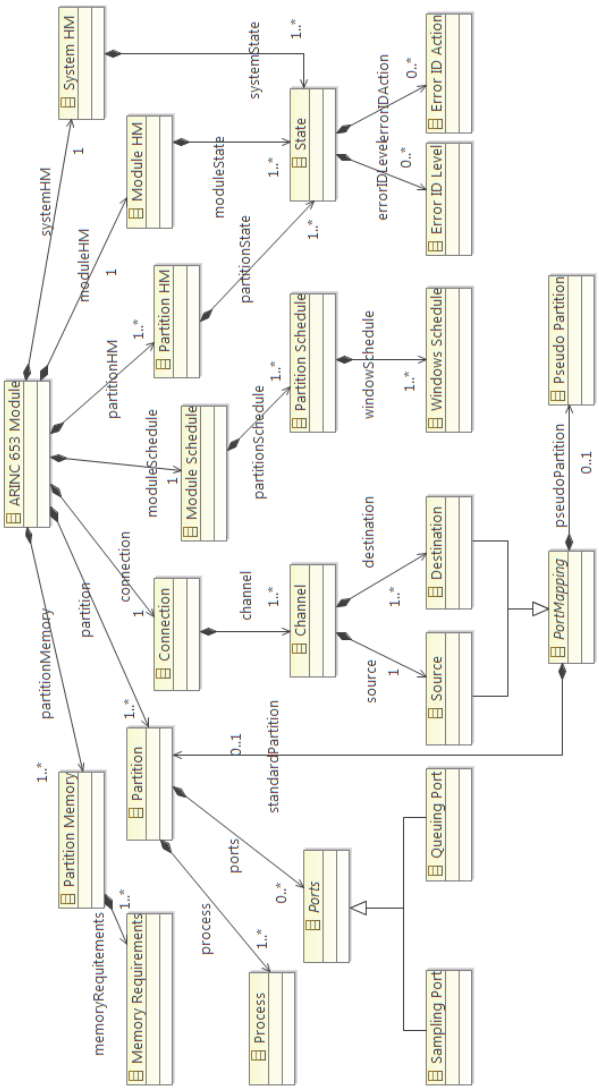


Figura A.1: Modelo do domínio

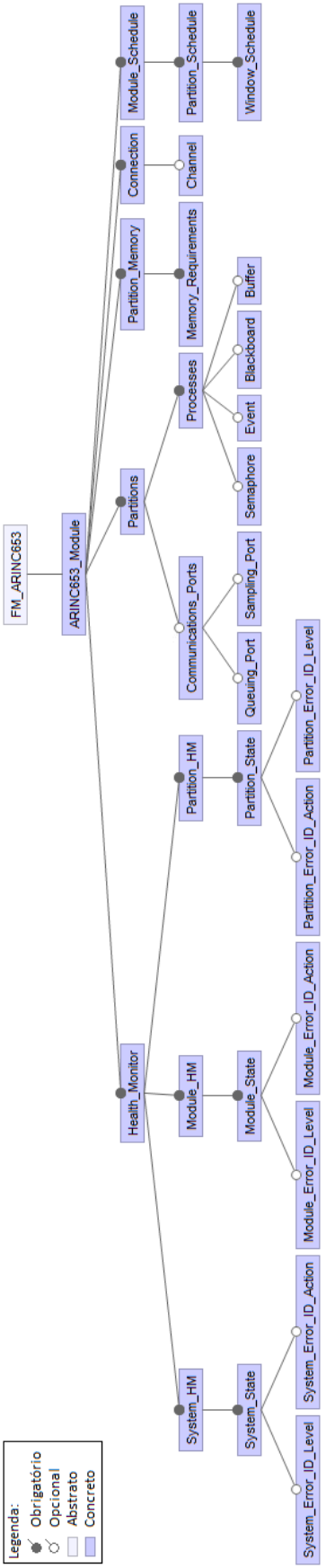


Figura A.2: *Feature model* do domínio

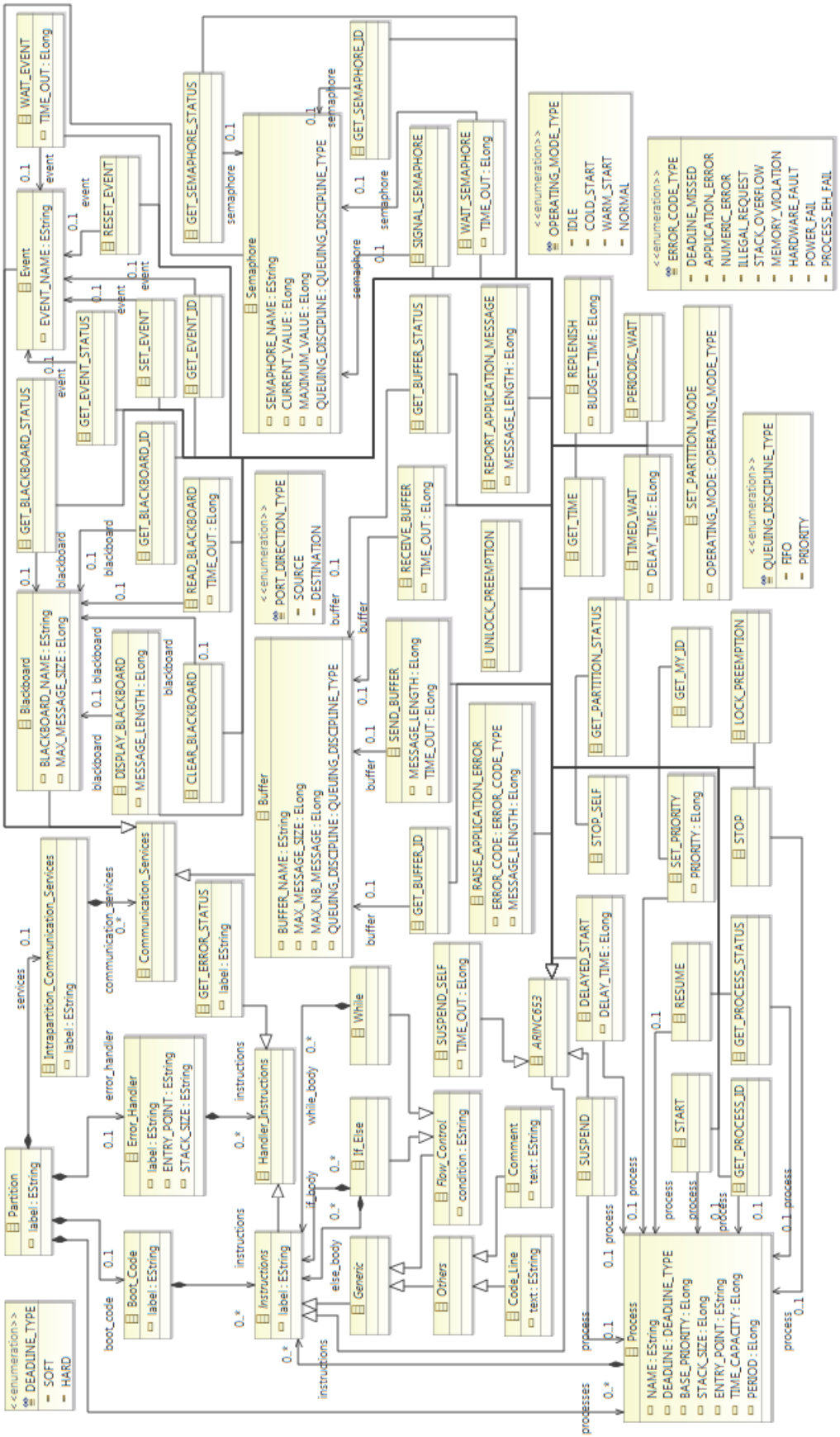


Figura A.3: Metamodelo dos conceitos comuns do domínio

Questionário de Recrutamento

1. Quais as linguagens de programação que se encontra familiarizado(a)? (Uma ou mais respostas)

Resposta:

☐ ADA ☐ C ☐ C++ ☐ JAVA ☐ Pascal ☐ Outras: _____

2. Encontra-se familiarizado(a) com IMA e o Standard ARINC 653? (Só uma opção)

Resposta:

☐ SIM ☐ NÃO

3. Se respondeu SIM na pergunta 2, fez ou faz parte da equipa de desenvolvimento de algum projecto relacionado com IMA\Standard ARINC 653? (Só uma opção)

Resposta:

☐ SIM ☐ NÃO

4. Se respondeu SIM na pergunta 3, qual a complexidade desse(s) projecto(s)? (Só uma opção)

Explicitação da escala:

1 – Desenvolveu uma aplicação simples com recurso à API referida no Standard ARINC 653;

5 – Desenvolveu o sistema operativo e/ou componentes que forneçam os serviços da API.

Resposta:

Pouco complexo ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 Muito complexo

5. Utiliza ambientes gráficos ou ambientes integrados de desenvolvimento (IDE)? (Só uma opção)

Resposta:

☐ SIM ☐ NÃO

6. Encontra-se familiarizado(a) com linguagens gráficas, por exemplo UML? (Só uma opção)

Resposta:

☐ SIM ☐ NÃO

Figura A.4: Questionário de recrutamento

Questionário de Avaliação das Linguagens

1. Qual o nível de satisfação sobre o processo de avaliação das linguagens? (Só uma opção)

Resposta:

Não gostei ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 Gostei muito

2. Na avaliação que só utiliza C, qual o nível de confiança que a sua solução funciona correctamente? (Só uma opção)

Resposta:

Não confio ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 Confio muito

3. Na avaliação que utiliza a DSL + C, qual o nível de confiança que a sua solução funciona correctamente? (Só uma opção)

Resposta:

Não confio ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 Confio muito

4. A utilização da DSL facilitou o desenvolvimento da sua solução? (Só uma opção)

Resposta:

Não facilitou ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 Facilitou muito

5. Qual das abordagens achou melhor? (Só uma opção)

Resposta:

☐ Só C ☐ DSL + C

6. Qual das abordagens foi tecnicamente mais difícil de concretizar? (Só uma opção)

Resposta:

☐ Só C ☐ DSL + C

Figura A.5: Questionário da primeira sessão (Página 1)

7. Na eventualidade de lhe ser pedido para desenvolver o código de uma partição em cada uma das seguintes situações, qual das opções escolheria? (Só uma opção)

a) Uma partição *aviónica* com dois processos. Um destes processos seria de sistema (responsável pela inicialização dos restantes) e outro em ciclo infinito que recebe informação de duas *queuing ports* e calcula um valor usando uma expressão auxiliar de forma a submeter o resultado numa *sampling port*. Tanto as *queuing ports* como a *sampling port* recebem e enviam a informação para outras partições.

Resposta:

☐ Só C ☐ DSL + C

b) Uma partição *aviónica* com diversos processos. Alguns destes processos comunicam entre si usando *Blackboards*, *Buffers*, *Events*, *Semaphores*. Adicionalmente é também usado *sampling ports* e *queuing ports* para comunicar com outras partições e pseudo partições.

Resposta:

☐ Só C ☐ DSL + C

c) Uma partição *aviónica* com diversos processos e responsável por operações complexas (ou seja, extensa com avultado número de linhas de código). Alguns destes processos comunicam entre si usando *Blackboards*, *Buffers*, *Events*, *Semaphores*. Usa-se *sampling ports* e *queuing ports* para comunicar com outras partições. Adicionalmente também são usadas outras operações disponíveis na API do Standard ARINC 653.

Resposta:

☐ Só C ☐ DSL + C

8. Algum dos seguintes ícones usados na DSL achou difícil perceber o significado?
(Uma ou mais respostas)

Resposta:

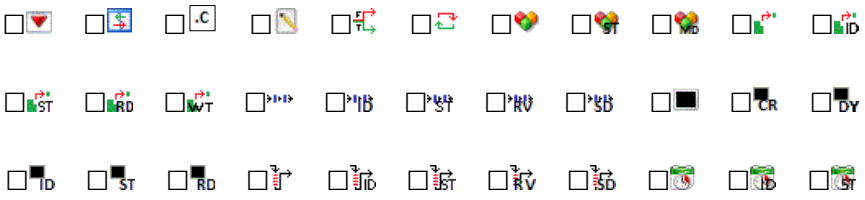
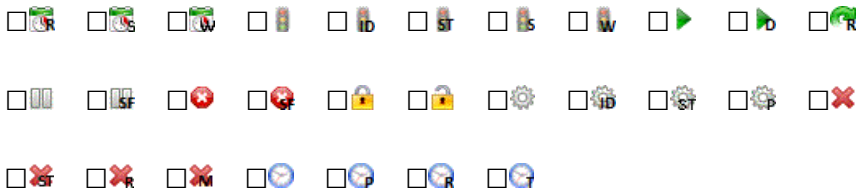


Figura A.6: Questionário da primeira sessão (Página 2)



9. Qual a classificação que atribui à facilidade em usar a DSL ? (Só uma opção)

Resposta:

Difícil ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 Fácil

10. Qual a classificação que atribui à confiança na obtenção de um bom resultado final quando usa a DSL ? (Só uma opção)

Resposta:

Não confio ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 Confio muito

11. A DSL exprime correctamente os conceitos do domínio, ou seja, se é expressiva? (Só uma opção)

Resposta:

Não exprime correctamente ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 Exprime correctamente

12. O que gostou menos na DSL?

Resposta:

_____
_____

13. O que gostou mais na DSL?

Resposta:

_____
_____

Figura A.7: Questionário da primeira sessão (Página 3)

Questionário de Avaliação da Linguagem

1. Qual o nível de satisfação sobre este segundo processo de avaliação da DSL? (Só uma opção)

Resposta:

Não gostei ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 Gostei muito

2. Qual o nível de confiança que no esqueleto da solução por si produzida usando a DSL? (Só uma opção)

Resposta:

Não confio ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 Confio muito

3. Qual a classificação que atribui à facilidade em usar a DSL ? (Só uma opção)

Resposta:

Difícil ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 Fácil

4. Qual a classificação que atribui à confiança na obtenção de um bom resultado final quando usa a DSL ? (Só uma opção)

Resposta:

Não confio ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 Confio muito

5. A DSL exprime correctamente os conceitos do domínio, ou seja, se é expressiva? (Só uma opção)

Resposta:

Não exprime correctamente ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 Exprime correctamente

Figura A.8: Questionário da segunda sessão (Página 1)

6. Tendo em consideração a DSL agora apresentada e em comparação com a DSL apresentada no primeiro exercício (realizado na semana anterior), identifique qual a afirmação que mais se apropria. (Só uma opção)

Resposta:

☐ A DSL agora apresentada demonstra um **menor nível** de usabilidade (mais difícil de utilizar);

☐ A DSL agora apresentada demonstra o **mesmo nível** de usabilidade;

☐ A DSL agora apresentada demonstra um **maior nível** de usabilidade (mais fácil de utilizar).

Figura A.9: Questionário da segunda sessão (Página 2)

12-09-2014

DISSERTAÇÃO DE MESTRADO EM ENGENHARIA INFORMÁTICA

Ricardo Alves N.39828

FAMÍLIA DE DSLS PARA *IMA*

AVALIAÇÃO DE USABILIDADE

FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE NOVA DE LISBOA

gmv INNOVATING SOLUTIONS

AGENDA

- *Integrated Modular Avionics* e Standard ARINC 653
- ARINC 653 (Definição)
- Componentes do ARINC 653 (Essenciais para a Avaliação de usabilidade da DSL)
- Apresentação da DSL
- Exemplo
- Exercício

INTEGRATED MODULAR AVIONICS E STANDARD ARINC 653

Figura 1: Airbus A380

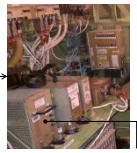


Figura 2: Baía *aviónica*

Figura 3: Arquitetura interna do módulo de computação

STANDARD ARINC 653

- Especifica o ambiente de operação base das aplicações avionicas usadas em IMA.
- Define uma API de uso geral entre o sistema operativo e as aplicações aviónicas (denominada APEX).
- Define os dados trocados estaticamente (via configuração) ou dinamicamente (via serviços) e o comportamento dos serviços fornecidos pelo sistema operativo.

COMPONENTES DO ARINC 653

- **Partição** – Conceito abstracto de alto nível. É onde residem as funções aviónicas num modulo particionado de acordo com restrições de espaço e memória;
- **Processo** – Componentes de uma partição que combinados dinamicamente fornecem as funcionalidades de uma partição;
- **Mecanismos de comunicação entre Partições** – Permite a comunicações em partições;
- **Mecanismos de comunicação dentro das Partições** – Permite a comunicações entre processos pertencentes à mesma partição;

APRESENTAÇÃO DA DSL:

ESCOLHER A PARTIÇÃO A DESENVOLVER

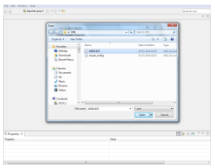


Figura 4: Escolha do ficheiro XML com a configuração do Módulo

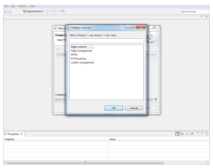


Figura 5: Escolha da partição a desenvolver

1

Figura A.10: Slides da primeira sessão (Slide 1 a 6)

APRESENTAÇÃO DA DSL: INTERFACE GRÁFICA

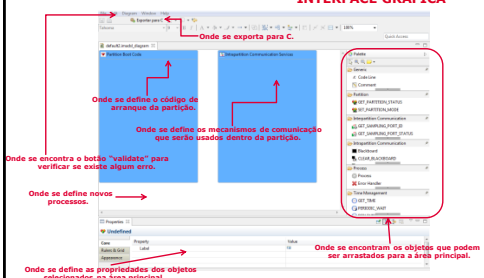


Figura 6: Interface Gráfica da DSL

EXEMPLO

Aplicação aviónica com dois processos:

- O primeiro processo é responsável pela inicialização da partição;
- O segundo processo é responsável continuamente (em ciclo) por:
  - Ler o valor de duas QUEUING PORTs (Stat_2Dq e Stat_3Dq);
  - Efetuar o somatório do valor inteiro obtido através das QUEUING PORTs (Stat_2Dq e Stat_3Dq);
  - Enviar o resultado do somatório obtido através da QUEUING PORT (Stat_4Dq).

EXEMPLO

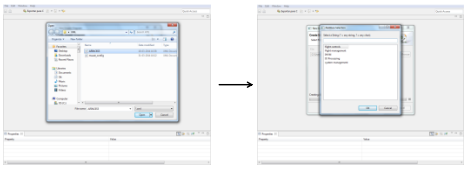


Figura 7: Escolha do ficheiro XML com a configuração do Módulo

Figura 8: Escolha da partição a desenvolver

EXEMPLO

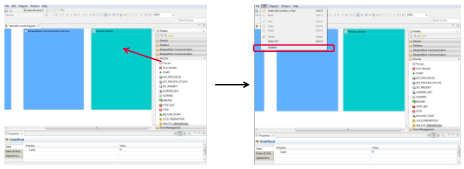


Figura 9: Criar o processo "Sensors"

Figura 10: Validar as construções efetuadas

EXEMPLO

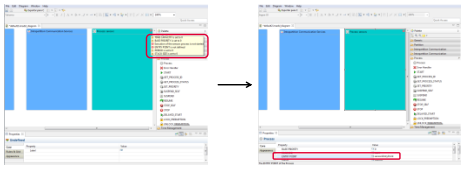


Figura 11: Visualização dos erros encontrados

Figura 12: Corrigir erro referente ao "ENTRY POINT"

EXEMPLO

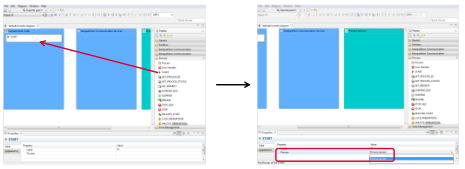


Figura 13: Corrigir erro referente ao arranque do processo "sensors", 1º passo

Figura 14: Selecionar o processo que será arrancado, 2º passo

Figura A.11: Slides da primeira sessão (Slide 7 a 12)

12-09-2014

EXEMPLO

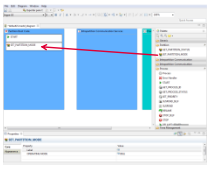


Figura 15: Arrastar a instrução para colocar a partição *em modo "NORMAL", 1º passo*

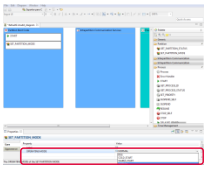


Figura 16: Selecionar o modo "NORMAL", 2º passo

EXEMPLO

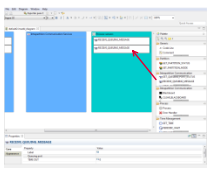


Figura 17: Arrastar a instrução para receber a mensagem

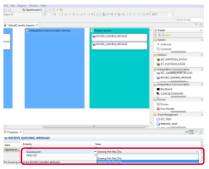


Figura 18: Selecionar a porta correta

EXEMPLO

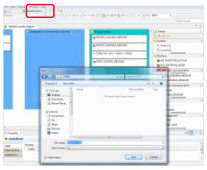


Figura 19: Gerar código C

Figura 20: Visualizar código gerado

EXERCÍCIO

Aplicação aviónica com três processos:

- O primeiro processo é responsável pela inicialização da partição;
- O segundo processo é responsável continuamente (em ciclo) por:
  - Ler o valor de uma QUEUING PORT (Stat_2Dq);
  - Escrever o valor lido num blackboard (Criar com o nome: BB1).
- O terceiro processo é responsável continuamente (em ciclo) por:
  - Ler o valor do blackboard (BB1);
  - Escrever o valor lido do blackboard (BB1) numa variável local.

AVALIAÇÃO

Aplicação aviónica com três processos:

- O primeiro processo é responsável pela inicialização da partição;
- O segundo processo é responsável continuamente (em ciclo) por:
  - Receber informação da QUEUING PORT (Stat_2Dq);
  - Enviar para outra partição através de uma QUEUING PORT (Stat_3Dq) a mensagem recebida anteriormente da QUEUING PORT (Stat_2Dq);
  - Aguardar até que o terceiro processo esteja livre (sincroniza).
- O terceiro processo é responsável continuamente (em ciclo) por:
  - Receber informação da QUEUING PORT (Stat_4Dq);
  - Executar a função msg alg(*msg);
  - Enviar pela QUEUING PORT (Stat_5Dq) o retorno da função msg alg(*msg);
  - Informar o segundo processo que terminou a tarefa (sincroniza).

Obrigado

GMVIS SKYSOFT, SA
Torre Fernão de Magalhães
Av. D. João II Lote 1.17.02, 7º Andar
1998 - 025 Lisboa Portugal
Tel. +351 21 382 93 66
Fax +351 21 386 64 93

RICARDO ALVES

E-MAIL (ACADÉMICO):
RJF.ALVES@CAMPUS.FCT.UNL.PT
E-MAIL (EMPRESARIAL):
RICARDO.ALVES@GMV.COM

1

Figura A.12: Slides da primeira sessão (Slide 13 a 18)

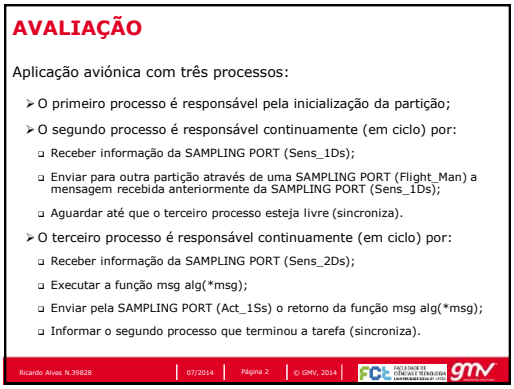


Figura A.13: Slides da segunda sessão (Slide 1 a 3)

Manual de utilizador

- Apresentação da principal interface gráfica da DSL:

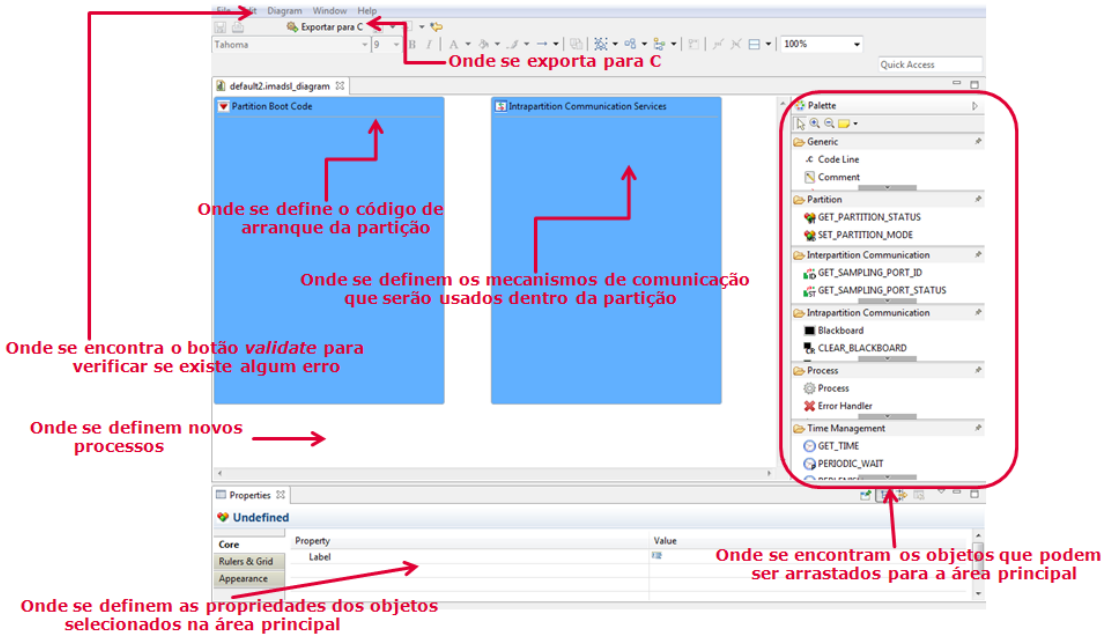


Figura 1: Principal interface gráfica da DSL

Na figura 1 apresenta-se a principal interface gráfica da DSL. Através desta interface define-se a estrutura da aplicação *aviónica* clicando nos elementos da *Palette* (à direita) e de seguida clicando na área principal de trabalho (centro e esquerda).

Na área principal de trabalho é possível criar elementos como o *process* e o *error handler* onde existe um espaço vazio utilizando a técnica de cliques referido no parágrafo anterior.

Os elementos *Partition Boot Code* e *Intrapartition Communication Services* não podem ser apagados. O primeiro elemento é onde se define o código de arranque da partição, que normalmente é responsável pelo arranque dos processos onde se irá implementar a lógica da aplicação *aviónica* a desenvolver. O segundo elemento é onde se define que serviços de comunicação e sincronização entre processos podem ser utilizados, tais como *event*, *semaphore*, *blackboard* e *buffer*.

Na zona inferior do ecrã é possível alterar as propriedades do elemento que estiver selecionado na área principal de trabalho.

Sempre que o utilizador desejar poderá validar as construções que realizou, através da opção *validate* no menu *edit*.

Finalmente, é possível exportar a aplicação para código C, clicando no botão existente na barra de ferramentas.

Figura A.14: Manual de utilizador (Página 1)

- Da figura 2 à figura 8 apresenta-se uma sequência de interfaces gráficas onde se demonstram os principais passos para produzir uma aplicação *aviónica* utilizando a DSL.

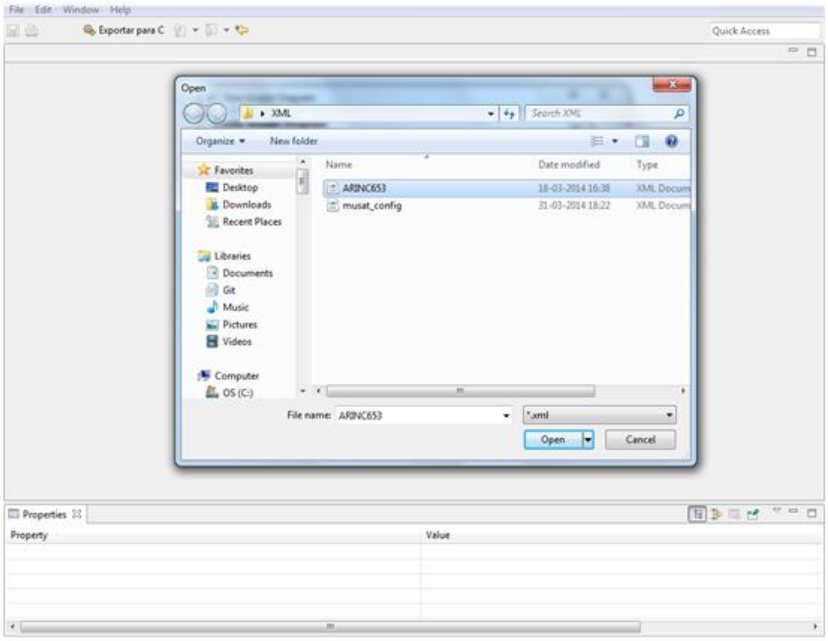


Figura 2: Escolher o ficheiro XML com a configuração do Módulo

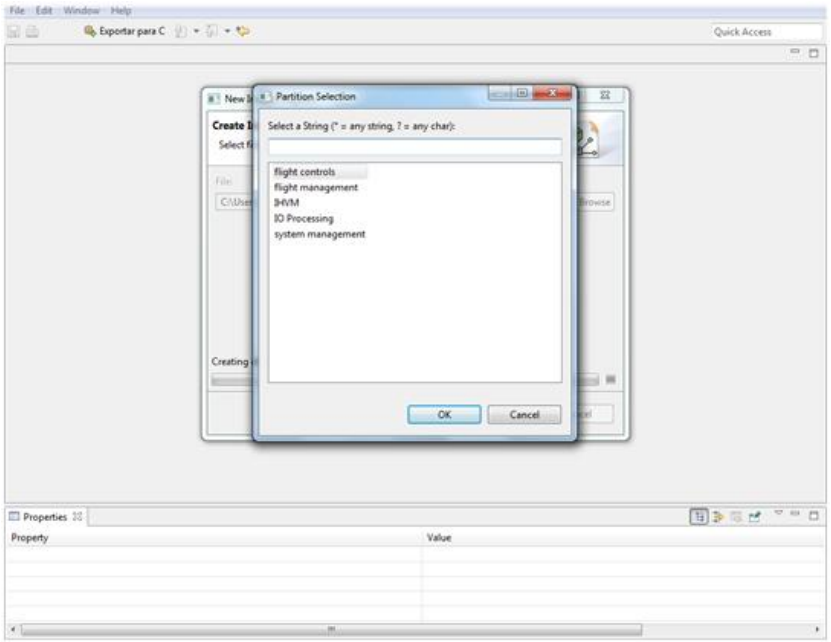


Figura 3: Escolher a partição a desenvolver

Figura A.15: Manual de utilizador (Página 2)

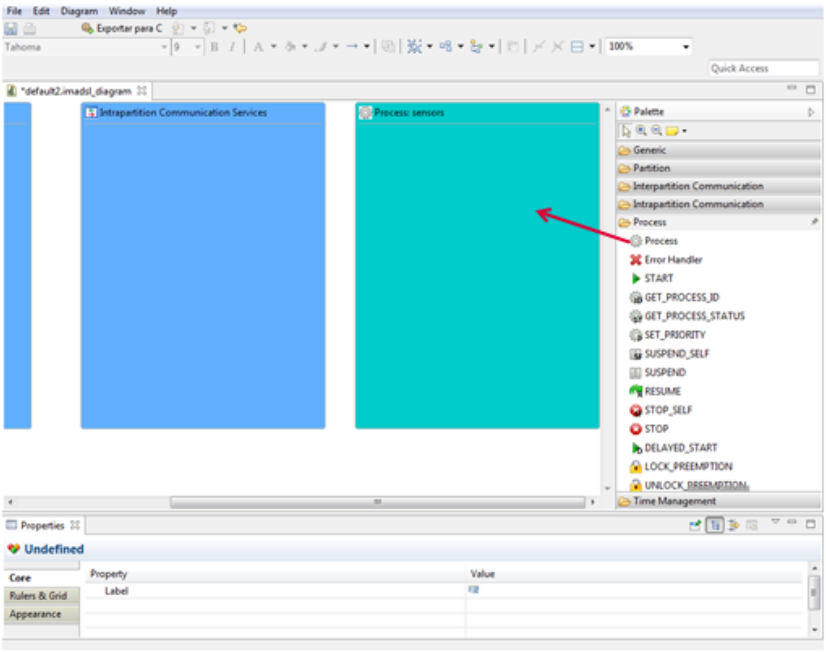


Figura 4: Criar elementos na área principal de trabalho

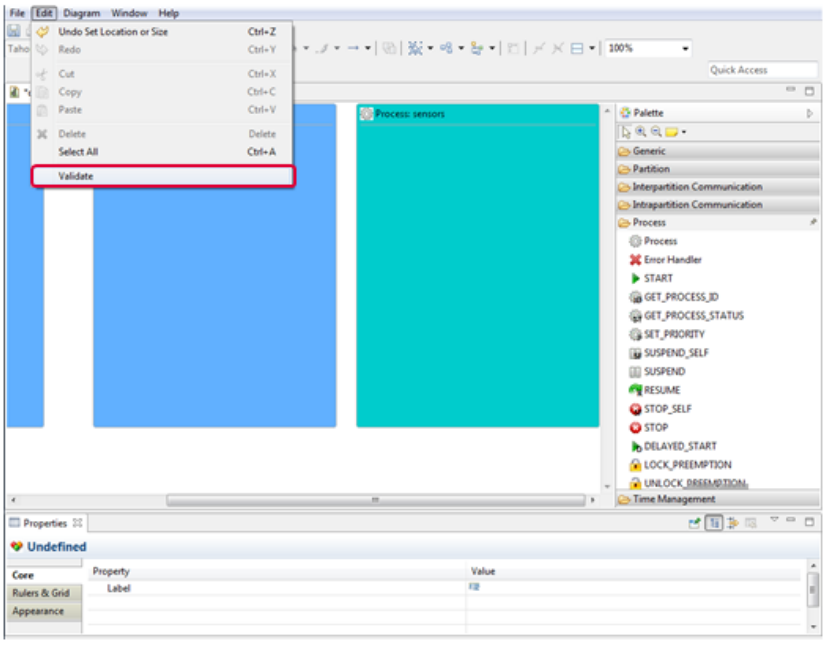


Figura 5: Validar as construções efetuadas

Figura A.16: Manual de utilizador (Página 3)

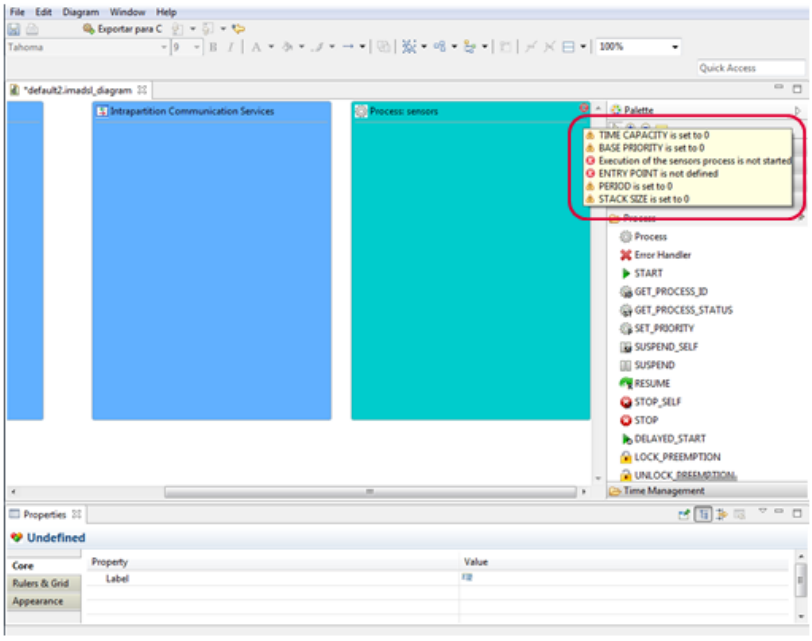


Figura 6: Visualizar os erros encontrados quando se coloca o cursor do rato por cima da indicação

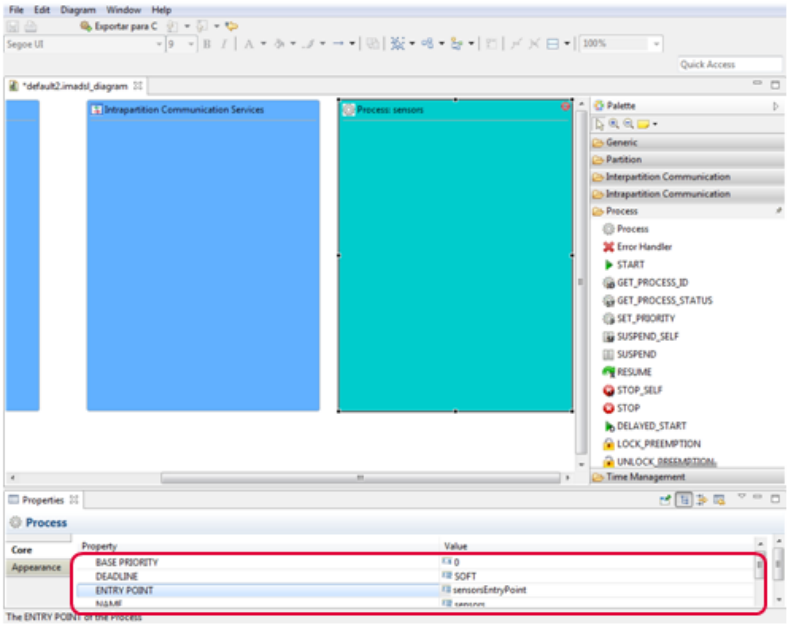


Figura 7: Alterar as propriedades do elemento selecionado na área principal de trabalho

Figura A.17: Manual de utilizador (Página 4)

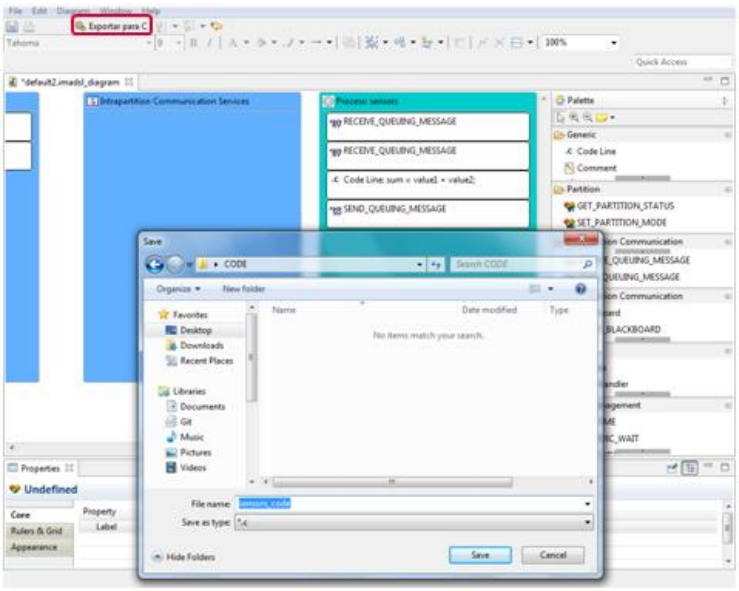


Figura 8: Gerar código C

Figura A.18: Manual de utilizador (Página 5)

```
  /**
   * Aplicação aviónica simples com dois processos:
   * 1) O primeiro processo é responsável pela inicialização da partição;
   * 2) O segundo processo é responsável continuamente (em ciclo) por:
   *  a) Ler o valor de duas QUEUING PORTs (Stat_2Dq e Stat_3Dq);
   *  b) Efetuar o somatório do valor inteiro obtido através das QUEUING PORTs (Stat_2Dq e Stat_3Dq);
   *  c) Enviar o resultado do somatório obtido através da QUEUING PORT (Stat_4Dq).
   *
   * O seguinte código não pertence a uma aplicação aviónica real, trata-se de um pequeno exemplo para
   * demonstrar o código produzido pela DSL, bem como, o reduzido número de linhas de código que é
   * necessário o utilizador desenvolver.
   *
   * Neste exemplo só foi necessário o utilizador escrever o seguinte código:
   *  int *queuingPort3_Msg = (int)*queuingPort1_Msg + (int)*queuingPort2_Msg;
   ***/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <a653.h>

void sum_entry_point(){

  /**
   * Variables Of QUEUING PORT: Stat_2Dq
   ***/
  MESSAGE_ADDR_TYPE queuingPort1_Msg;
  MESSAGE_SIZE_TYPE queuingPort1_Length;
  QUEUING_PORT_ID_TYPE queuingPort1_Id;
  QUEUING_PORT_STATUS_TYPE queuingPort1_Status;
  RETURN_CODE_TYPE queuingPort1_ReturnCode;
  /**
   * Variables Of QUEUING PORT: Stat_3Dq
   ***/
  MESSAGE_ADDR_TYPE queuingPort2_Msg;
  MESSAGE_SIZE_TYPE queuingPort2_Length;
  QUEUING_PORT_ID_TYPE queuingPort2_Id;
  QUEUING_PORT_STATUS_TYPE queuingPort2_Status;
  RETURN_CODE_TYPE queuingPort2_ReturnCode;
  /**
   * Variables Of QUEUING PORT: Stat_4Dq
   ***/
  MESSAGE_ADDR_TYPE queuingPort3_Msg;
  MESSAGE_SIZE_TYPE queuingPort3_Length;
  QUEUING_PORT_ID_TYPE queuingPort3_Id;
  QUEUING_PORT_STATUS_TYPE queuingPort3_Status;
  RETURN_CODE_TYPE queuingPort3_ReturnCode;

  while(true) { //TODO: Verify WHILE condition

    GET_QUEUING_PORT_ID("Stat_2Dq", &queuingPort1_Id, &queuingPort1_ReturnCode);
    if (queuingPort1_ReturnCode != NO_ERROR) {
      printf("Couldn't obtain the queuing port id: %d\n", queuingPort1_ReturnCode);
    }

    GET_QUEUING_PORT_ID("Stat_3Dq", &queuingPort2_Id, &queuingPort2_ReturnCode);
    if (queuingPort2_ReturnCode != NO_ERROR) {
      printf("Couldn't obtain the queuing port id: %d\n", queuingPort2_ReturnCode);
    }

    GET_QUEUING_PORT_ID("Stat_4Dq", &queuingPort3_Id, &queuingPort3_ReturnCode);
    if (queuingPort3_ReturnCode != NO_ERROR) {
      printf("Couldn't obtain the queuing port id: %d\n", queuingPort3_ReturnCode);
    }

    RECEIVE_QUEUING_MESSAGE(queuingPort1_Id, 0, (MESSAGE_ADDR_TYPE)queuingPort1_Msg, &queuingPort1_Length,
&queuingPort1_ReturnCode);
    if (queuingPort1_ReturnCode != NO_ERROR) {
      printf("Couldn't receive the queuing message: %d\n", queuingPort1_ReturnCode);
    }

    RECEIVE_QUEUING_MESSAGE(queuingPort2_Id, 0, (MESSAGE_ADDR_TYPE)queuingPort2_Msg, &queuingPort2_Length,
&queuingPort2_ReturnCode);
    if (queuingPort2_ReturnCode != NO_ERROR) {
      printf("Couldn't receive the queuing message: %d\n", queuingPort2_ReturnCode);
    }

    //TODO: CODE LINE
    int *queuingPort3_Msg = (int)*queuingPort1_Msg + (int)*queuingPort2_Msg;
    //End CODE LINE

    SEND_QUEUING_MESSAGE(queuingPort3_Id, (MESSAGE_ADDR_TYPE)queuingPort3_Msg, 0, 0, &queuingPort3_ReturnCode);
    if (queuingPort3_ReturnCode != NO_ERROR) {
      printf("Couldn't send the queuing message: %d\n", queuingPort3_ReturnCode);
```

Figura A.19: Código de uma aplicação *aviónica* (Página 1)

```
    }

  }
}

int entry_point(void) {
  /**
   * Variables Of QUEUING PORT: Stat_2Dq
   ***/
  MESSAGE_ADDR_TYPE queuingPort1_Msg;
  MESSAGE_SIZE_TYPE queuingPort1_Length;
  QUEUING_PORT_ID_TYPE queuingPort1_Id;
  QUEUING_PORT_STATUS_TYPE queuingPort1_Status;
  RETURN_CODE_TYPE queuingPort1_ReturnCode;
  /**
   * Variables Of QUEUING PORT: Stat_3Dq
   ***/
  MESSAGE_ADDR_TYPE queuingPort2_Msg;
  MESSAGE_SIZE_TYPE queuingPort2_Length;
  QUEUING_PORT_ID_TYPE queuingPort2_Id;
  QUEUING_PORT_STATUS_TYPE queuingPort2_Status;
  RETURN_CODE_TYPE queuingPort2_ReturnCode;
  /**
   * Variables Of QUEUING PORT: Stat_4Dq
   ***/
  MESSAGE_ADDR_TYPE queuingPort3_Msg;
  MESSAGE_SIZE_TYPE queuingPort3_Length;
  QUEUING_PORT_ID_TYPE queuingPort3_Id;
  QUEUING_PORT_STATUS_TYPE queuingPort3_Status;
  RETURN_CODE_TYPE queuingPort3_ReturnCode;
  /**
   * Variables Of PROCESS: sum
   ***/
  PROCESS_ID_TYPE process1_Id;
  RETURN_CODE_TYPE process1_ReturnCode;
  PROCESS_STATUS_TYPE process1_Status;

  PROCESS_ATTRIBUTE_TYPE atts;

  CREATE_QUEUING_PORT("Stat_2Dq", 30, 30, DESTINATION, FIFO, &queuingPort1_Id, &queuingPort1_ReturnCode);
  if (queuingPort1_ReturnCode != NO_ERROR) {
    printf("Couldn't create queuing port '%s': %d\n", queuingPort1_Id, queuingPort1_ReturnCode);
  }

  CREATE_QUEUING_PORT("Stat_3Dq", 30, 30, DESTINATION, FIFO, &queuingPort2_Id, &queuingPort2_ReturnCode);
  if (queuingPort2_ReturnCode != NO_ERROR) {
    printf("Couldn't create queuing port '%s': %d\n", queuingPort2_Id, queuingPort2_ReturnCode);
  }

  CREATE_QUEUING_PORT("Stat_4Dq", 30, 30, DESTINATION, FIFO, &queuingPort3_Id, &queuingPort3_ReturnCode);
  if (queuingPort3_ReturnCode != NO_ERROR) {
    printf("Couldn't create queuing port '%s': %d\n", queuingPort3_Id, queuingPort3_ReturnCode);
  }

  atts.PERIOD = 0;
  atts.TIME_CAPACITY = 0;
  atts.STACK_SIZE = 0;
  atts.BASE_PRIORITY = 0;
  atts.DEADLINE = SOFT;
  atts.ENTRY_POINT = &sum_entry_point;
  strcpy(atts.NAME, "SUM");

  CREATE_PROCESS(&atts, &process1_Id, &process1_ReturnCode);
  if (process1_ReturnCode != NO_ERROR) {
    printf("Couldn't create process '%s': %d\n", atts.NAME, process1_ReturnCode);
  }

  START(processId1, &returnCode);
  if (returnCode != NO_ERROR) {
    printf("Couldn't start process %ld: %d\n", processId1, returnCode);
  }

  SET_PARTITION_MODE(NORMAL, &returnCode);
  if (returnCode != NO_ERROR) {
    printf("Couldn't switch to NORMAL MODE: %d\n", returnCode);
  }

  return NO_ERROR;
}
```

Figura A.20: Código de uma aplicação *aviónica* (Página 2)