Universidade do Minho
Escola de Engenharia

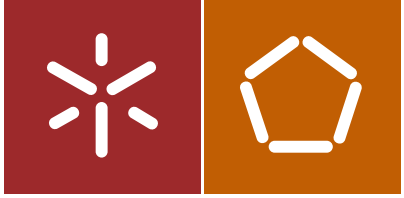
Eduardo Manuel Ferreira Domingues

Refactoring de um AUTOSAR-ENABLED RTOS: Modelação Segundo o Standard IP-XACT

Novembro de 2014

Eduardo Manuel Ferreira Domingues Refactoring de um AUTOSAR-ENABLED RTOS: Modelação Segundo o Standard IP-XACT

UMinho | 2014



Universidade do Minho
Escola de Engenharia

Eduardo Manuel Ferreira Domingues

Refactoring de um AUTOSAR-ENABLED RTOS: Modelação Segundo o Standard IP-XACT

Dissertação de Mestrado
Ciclo de Estudos Integrados Conducentes ao Grau de
Mestre em Engenharia Electrónica Industrial e Computadores

Trabalho efetuado sob a orientação do
Professor Doutor Adriano Tavares

Novembro de 2014

DECLARAÇÃO

Nome: Eduardo Manuel Ferreira Domingues

Correio electrónico: eduardo.m.f.domingues@gmail.com

TIm.: 914892750

Número do Bilhete de Identidade:13930561

Título da dissertação:

Refactoring de um AUTOSAR-ENABLED RTOS: Modelação Segundo o Standard IP-XACT

Ano de conclusão: 2014

Orientado:

Professor Doutor Adriano Tavares

Designação do Mestrado:

Ciclo de Estudos Integrados Conducentes ao Grau de Mestre em Engenharia Electrónica Industrial e Computadores

Área de Especialização:Sistemas Embebidos

Escola:Escola de Engenharia

Departamento: Electronica Industrial

Guimarães, ___/___/_____

Assinatura: _____

AGRADECIMENTOS

Em primeiro lugar gostaria de agradecer à minha família por todos os sacrifícios que fizeram para me dar as oportunidades e as condições para concluir esta importante etapa da minha formação académica.

Ao meu orientador Professor Doutor Adriano Tavares por todo o tempo investido, conselhos dados, e confiança e conhecimentos transmitidos, e também pela oportunidade de realizar esta dissertação.

Aos meus colegas de laboratório pela ajuda prestada em todos os momentos.

Aos meus verdadeiros amigos que sempre estiveram presentes nos momentos mais necessários, e no seu grande e indispensável apoio.

À minha namorada, pelo apoio, compreensão, carinho e paciência dedicada durante este processo.

RESUMO

Nos dias de hoje o uso de sistemas operativos em sistemas embebidos começa a ser cada vez mais comum. Contudo, para reduzir a pressão no *Time-To-Market* e consequentemente ganhar vantagens sobre a competição, os RTOS para sistemas embebidos devem ser modelados num nível de abstração mais elevado quando comparado com o nível de C/C++ usado atualmente.

Por isso os objetivos desta dissertação são (1) fazer o *refactoring* do AUTOSAR-ENABLED RTOS escolhido, neste caso o trampoline OS, de acordo com um modelo IP-XACT estendido que representa os IPs de *software* a serem executados na plataforma de *hardware* escolhida, e (2) integrá-lo num repositório IP-XACT usado para dar suporte ao desenvolvimento de sistemas baseado em modelos AUTOSAR.

Para o efeito foi implementada uma *framework* orientada a modelos e compatível com AUTOSAR para gerar e gerir IPs individuais e o próprio RTOS, que posteriormente serão referenciados como blocos básicos da *framework* ou ambiente de desenvolvimento AUTOSAR. Juntamente com os ficheiros de código C/C++ serão também gerados os ficheiros do projeto, da configuração e da aplicação, como por exemplo, (1) o ficheiro OIL que terá a configuração do RTOS, (2) o código C para a aplicação pretendida, e (3) um ficheiro XML com a descrição IP-XACT do RTOS desenhado.

Palavras-chave: AUTOSAR, RTOS, Trampoline, IP-XACT, IDE, Orientado a Modelos

ABSTRACT

Nowadays the use of operating systems for embedded systems begins to be increasingly common. However, to reduce the Time-to-Market pressure and consequently gain advantage over the competitors, RTOS for embedded systems should be modeled in a higher level of abstraction when compared with the C/C++ level used nowadays.

Therefore the objectives of this dissertation are (1) refactoring a chosen AUTOSAR-ENABLED RTOS, in this case the trampoline OS, according to an extended IP-XACT model that represent software IPs deployable on the chosen hardware platform, and (2) integrating it in a IP-XACT repository used to support model-based development of AUTOSAR systems.

In doing so, a model-based and AUTOSAR-enabled framework was implemented to generate and manage individual IPs and the RTOS itself, which later will be referenced as building blocks in the framework. Altogether, with C/C++ code files will be also generated, design, configuration and application files such as (1) OIL file providing RTOS configuration, (2) C code for the intended application, and (3) an XML file describing in IP-XACT model for the designed RTOS.

Keywords: AUTOSAR, RTOS, Trampoline, IP-XACT, IDE, Model-Based

ÍNDICE GERAL

ABREVIATURAS E SIGLAS

AUTOSAR – AUTomotive Open System Architecture

API – Application Programming Interface

DSL – Domain Specific Language

ECU – Engine Control Unit

EDA – Electronic Design Automation

IDE – Integrated Development Environment

I/O – Input/Output

IP – Intellectual Property

OIL – OSEK Implementation Language

OS – Operative System

OSEK – Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen ("Open Systems and the Corresponding Interfaces for Automotive Electronics")

RTOS – Real Time Operating System

SO – Sistema Operativo

SoC – System on Chip

VDX – Vehicle Distributed eXecutive

XML – Extensible Markup Language

ÍNDICE DE FIGURAS

ÍNDICE DE TABELAS

Capítulo 1

INTRODUÇÃO

Nos dias de hoje o uso de Sistemas Operativos em sistemas embebidos começa a ser cada vez mais comum. Contudo, para reduzir a pressão no *Time-To-Market* e consequentemente ganhar vantagens sobre a competição, os RTOS para sistemas embebidos devem ser modelados num nível de abstração mais elevado quando comparado com o nível de C/C++ usado atualmente.

Na indústria automóvel isto também é uma realidade. O aumento da eletrónica está cada vez mais presente nessa área. Os veículos mais modernos têm vários ECUs para fazer o controlo das várias partes do veículo, e para se poder ter a certeza que o sistema cumpre os requisitos funcionais, como isolar subsistemas e integrar componentes de *software* fornecidos por diferentes fornecedores, surgiu a necessidade de usar sistemas operativos nos ECUs dos veículos [1].

Ainda na indústria automóvel foi definido o *standard* AUTOSAR para especificar e assegurar como devem ser feitos os componentes de *software*, a definição do Sistema Operativo, e a compatibilidade do *software* entre fabricantes diferentes. O *standard* da AUTOSAR usa um *standard* já existente, o OSEK OS, para definir os seus parâmetros para o sistema operativo [2].

Fazendo o *refactoring* do sistema operativo dividindo o seu código em módulos simplifica a sua compreensão e facilita qualquer alteração que se queira fazer ao mesmo. A modelação feita seguindo o *standard* IP-XACT permite a integração do RTOS em plataformas de desenvolvimento que sigam o mesmo *standard*, fazendo assim com que seja mais fácil de o usar e não tenha que ser criada uma plataforma específica para um RTOS específico.

Nesta dissertação, pretende-se que se faça o *refactoring* de um sistema operativo para a indústria automóvel seguindo uma metodologia orientada a modelos com posterior integração

numa plataforma de desenvolvimento. Para isso será escolhido um sistema operativo que já tenha sido construído seguindo o *standard* OSEK.

1.1. Objetivos

O objetivo principal desta dissertação é fazer o *refactoring* do AUTOSAR-ENABLED RTOS escolhido, neste caso o trampoline, e integrá-lo num repositório, com modelação segundo o *standard* IP-XACT, de uma plataforma de design de *hardware* e *software*.

Para o integrar num repositório, o mais simples é gerar o IP do mesmo segundo o *standard* IP-XACT, visto que o repositório está preparado para o mesmo *standard*.

Para cumprir os objetivos é necessário construir um IDE para gerar o IP e o próprio sistema operativo. Esse IDE vai permitir fazer um desenho do sistema, uma representação usando blocos e ligações para representar o sistema pretendido. Com o desenho do mesmo o utilizador procederá à configuração de cada componente individualmente para que o mesmo corresponda ao desejado pelo utilizador.

O IDE terá também de fazer a verificação e validação do código, para que corresponda a todas as regras definidas, procedendo posteriormente à geração de código.

Na geração de código é esperado que, para cada sistema operativo desenhado, sejam gerados três ficheiros. O primeiro ficheiro será do tipo OIL e terá a configuração do sistema operativo. O segundo ficheiro será código C com o código da aplicação. E por fim, um ficheiro XML com a descrição em IP-XACT do sistema operativo desenhado.

1.2. Motivação

A escolha do tema da presente dissertação baseou-se nos conhecimentos adquiridos no âmbito da especialização em sistemas embebidos do curso de Engenharia Eletrónica Industrial e de

Computadores, e também pelo interesse na área de engenharia de desenvolvimento de *software*. Esta área encontra-se cada vez mais em desenvolvimento na sociedade atual, que está cada vez mais voltada para o uso de aplicações de *software* que permitam executar várias tarefas em simultâneo, além de facilitarem as mesmas tarefas.

É também pretendido aumentar o conhecimento teórico e prático inerente à construção de ambientes de desenvolvimento de *software*, IDEs, aplicando ao mesmo tempo esses conhecimentos num projeto prático que permite uma demonstração mais real dos mesmos.

Trata-se também de um tema de grande curiosidade pessoal para o autor da dissertação, pois permite o estudo aprofundado de ferramentas de desenvolvimento de IDEs e de *software* para a indústria automóvel.

1.3. Estrutura da Dissertação

A presente dissertação encontra-se dividida em sete capítulos, sendo eles: a introdução, o estado da arte, fundamentação teórica, especificação do sistema, a implementação, testes e resultados, e a conclusão.

O primeiro capítulo, "Introdução", é a inicialização da dissertação, no qual é realizado um enquadramento ao tema desta dissertação, apresentados os objetivos propostos, os motivos que levarão à execução desta dissertação e a estrutura da mesma.

O segundo capítulo, "Estado da Arte", encontra-se uma pena referencia ao que já existe no âmbito dos ambientes integrados de desenvolvimento.

O terceiro capítulo, "Fundamentação Teórica", é explicado a teoria estudada para o desenvolvimento desta dissertação.

O quarto capítulo, "Especificação do Sistema", constitui uma análise geral do sistema, e serão abordadas todas as restrições, validações e verificações, assim como todas as funcionalidades que o IDE deve proporcionar.

Ao longo do quinto capítulo, “Implementação”, é exposto o processo de implementação do sistema.

No sexto capítulo, “Testes e Resultados”, são expostos os resultados experimentais obtidos.

Por fim, no sétimo capítulo, “Conclusão”, serão explicadas as conclusões da dissertação, e algumas propostas para trabalhos futuros.

Capítulo 2

ESTADO DA ARTE

Neste capítulo será abordado o conceito de IDE (ambiente integrado de desenvolvimento) e a sua importância. Também serão apresentados alguns dos IDEs mais utilizados atualmente. Por fim será apresentada a escolha do autor.

2.1. IDE (Ambiente Integrado de Desenvolvimento)

A sigla IDE significa *Integrated Development Environment*, que se traduz para Ambiente Integrado de Desenvolvimento. Um IDE é uma aplicação de *software* que contém um conjunto de ferramentas para a construção de aplicações. Atualmente, a maioria dos IDEs tem um ambiente gráfico para a construção de aplicações e são compostos por algumas ferramentas base [3], que são:

- Um editor de código;
- Um compilador;
- Um *debugger;*
- Em alguns casos um construtor de GUI (Interface Gráfica para o Utilizador).

Vários dos IDEs mais modernos têm ainda mais ferramentas de desenvolvimento integradas, que permitam que vários programadores trabalhem no mesmo código. Já existiam esse tipo de ferramentas isoladas, mas agora existem *plugins* para alguns dos IDEs mais usados [4].

Com o uso de IDEs é possível automatizar a produção de *software*, o que aumenta a produtividade mesmo com o uso de uma menor mão-de-obra, diminuindo assim os custos.

De acordo com [5] os IDEs mais compensadores para programadores profissionais em 2013 e as linguagens mais usadas encontram-se na tabela seguinte:

Tabela 1 – IDEs mais usados 2013 e linguagens de programação

IDES	LINGUAGENS DE PROGRAMAÇÃO
VISUAL STUDIO	C++, C#, VB
NETBEANS	Java,C/C++, Ruby, HTML5, PHP
ECLIPSE	Java
CODE::BLOCKS	C/C++
APTANA STUDIO 3	PHP, Ruby, Pythom

2.2. Importância do IDE

O uso de IDEs é importante na construção de aplicações de *software* pois o IDE é desenhado para maximizar a produtividade do programador[6], visto que têm todas as ferramentas necessárias para o desenvolvimento da aplicação disponíveis para o utilizador com interfaces muito semelhantes para uma fácil utilização.

IDEs, como é o caso do Visual Studio, permitem criar outros IDEs que mais tarde podem ser usados como *plugins*. O uso de IDEs diminui a probabilidade de erros de sintaxe e ajuda a sua correção durante a produção do código, além de ajudar a evitar *bugs* no instante de execução. Será feito nesta dissertação um IDE para gerar código C evitando assim *bugs* que poderiam ocorrer se fossem codificados manualmente.

Os IDEs permitem a reutilização de código, e no caso do IDE a ser construído, todo o código será gerado seguindo uma abordagem generativa a partir de artefactos ou IPs já testados e presentemente num repositório.

Como um IDE permite gerar código a partir da construção de modelos, ou chamada de bibliotecas já produzidas e testadas, facilita a produção de código complexo, tornando assim muito mais fácil gerar grandes e complexos códigos usando apenas um IDE. Assim sendo, o código de um sistema complexo poderá ser produzido por um menor número de profissionais.

Algumas características presentes nas ferramentas de IDEs e que ajudam na produção de código são [7]:

- Código colorido, uma característica muito importante pois facilita a leitura do código;
- Preenchimento automático do código que ajuda muito o programador, mostrando uma lista de todas as opções disponíveis;
- Procura de comandos únicos que podem ser usados numa situação específica;
- *Refactoring* que permite a restruturação do código, por exemplo, através de alteração de nomes de classes ou métodos, sem ter de se fazer uma pesquisa manual por incoerência em outros ficheiros;
- *Snippets* de código, que ajudam a evitar que haja código repetido;
- Sistema para gestão de versões, para permitir ter várias versões do código guardadas caso seja preciso fazer retrocessos no processo sem perder grande parte do código.

No Anexo A será apresentado brevemente a evolução dos IDEs.

2.3. Ferramentas para criar IDEs

Existem várias ferramentas diferentes para criar IDEs no mercado. Serão agora apresentadas algumas das mais usadas, assim como as suas funcionalidades, requisitos e o que as mesmas proporcionam.

2.3.1. Qt

O Qt é uma ferramenta que permite desenvolver aplicações multiplataforma, que permite desenvolver aplicações gráficas de *software* e tem uma execução nativa em C++ [8]. Permite trabalhar também em diferentes sistemas operativos, como é o caso de Linux, Mac OS, Windows, Embedded Linux, Symbian e outros mais [9]. Em cada ambiente diferente precisa de um compilador específico para esse sistema, compilador este capaz de compilar e gerar código executável para o sistema nativo onde a aplicação está instalada. Segue-se o exemplo de alguns compiladores usados em diferentes sistemas:

Tabela 2 – Compiladores para diferentes plataformas

PLATFORMA	COMPILADORES
LINUX	GCC 4.4
MICROSOFT WINDOWS	GCC 4.4 (MinGW), MSVC 2005/2008
MAC OS	GCC (Fornecido pela Apple)
EMBEDDED LINUX	GCC (Conforme a versão do sistema)
SYMBIAN (SYMBIAN/S60 5.0)	RVCT 2.2, WINSCW 3.2.5, GCCE

As funcionalidades que o Qt disponibiliza e a sua utilização proporcionam ao utilizador as seguintes vantagens [10]:

- Fácil e rápida adaptação ao Qt através do uso de *Wizards* para criar projetos, e o acesso a um repositório com projetos recentes e tutoriais;
- Editor gráfico integrado para criar aplicações mais apelativas ao utilizador, e de mais fácil utilização;
- O avançado editor de C++ para desenvolver aplicações, que fornece várias características, como coloração do texto, evitar repetições de código, e outras.
- Permite compilar e testar o código no ambiente de trabalho, e também enviar um executável compilado para outras plataformas, como por exemplo um sistema operativo Linux para um sistema embebido;
- Permite o *debug* com os *debuggers* GNU, CDB e LLDB usando um interface gráfico, ajudando a perceber melhor a estrutura das classes do Qt;
- Possui ferramentas para fazer validação e verificação de possível ocorrência de problemas de memória;
- Permite criar pacotes de instalação para várias plataformas;
- Fácil acesso a informação devido ao sistema de ajuda do Qt que é sensível ao contexto em que a ajuda é pedida.

Segundo [11] a Figura 1 é a ilustração básica de todos os módulos que compõem o IDE Qt.

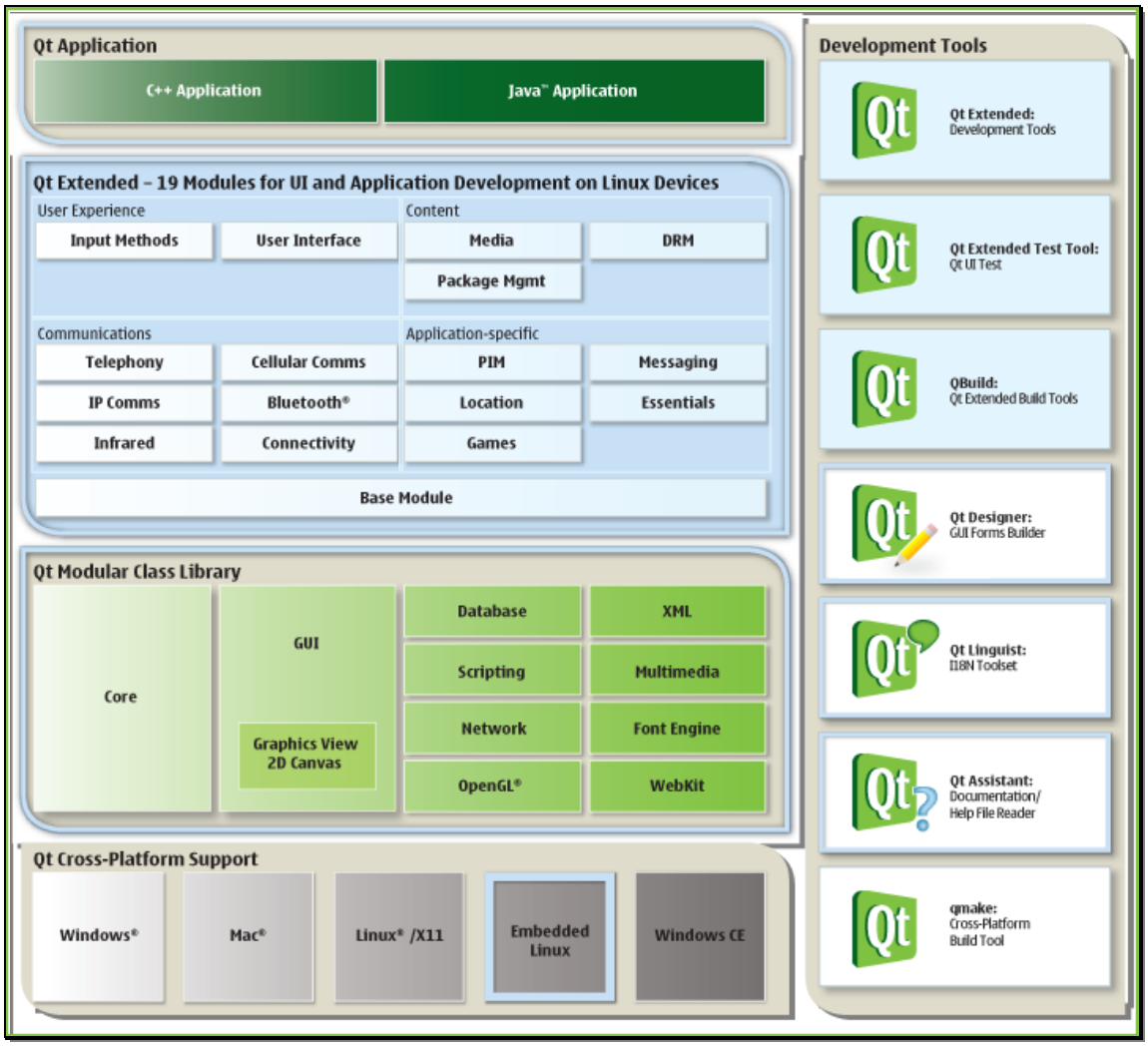


Figura 1 – Módulos que compõem o Qt

2.3.2. Eclipse

O IDE Eclipse foi desenvolvido em 2001 pela IBM e rapidamente se tornou num dos IDEs mais utilizados, principalmente devido ao facto de utilizar a biblioteca gráfica SWT (Standard Widget Toolkit) [12]. Trata-se de um IDE *open source*, cuja construção é totalmente baseada numa arquitetura *pure plugin*.

Como o Eclipse é principalmente utilizado para programar em linguagem Java, mas também oferece suporte a outras linguagens, como é o caso de C/C++, através do uso de *plugins*. É um IDE que permite o desenvolvimento multiplataforma, uma vez que todas as aplicações criadas

recorrem ao Eclipse para gerar um código intermédio que depois é interpretado e executado para *Java Virtual Machine*.

O Eclipse também fornece várias ferramentas para a construção de diferentes tipos de aplicação, e também fornece ferramentas para se integrar na maioria dos sistemas de desenvolvimento. Como já referido, é *open source* e grátis mas mesmo assim tem um bom suporte técnico. É um IDE extensível, por causa dos *plugins*, e configurável [13].

Como o Eclipse é baseado em *pluings* também apresenta desvantagens, principalmente para utilizadores iniciantes, pois o processo de instalação pode tornar-se confuso, dependendo da quantidade de *plugins* necessários para desenvolver a aplicação desejada.

Segundo [12] a figura seguinte apresenta a arquitetura do Eclipse, e evidencia que a sua estrutura se baseia no uso de *plugins* (Figura 2).

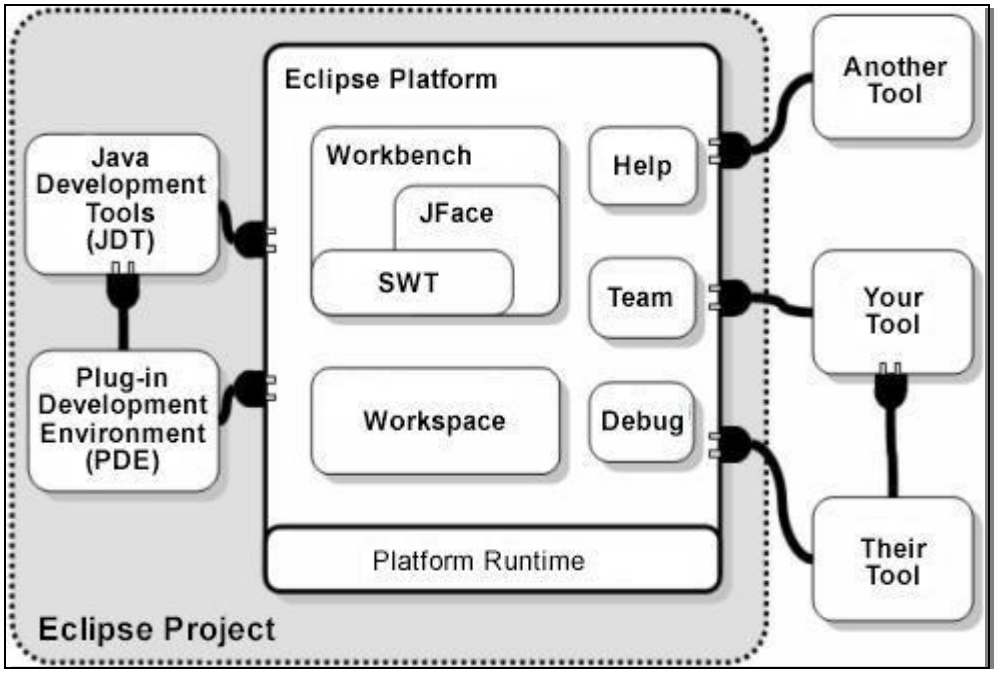


Figura 2 – Arquitetura do Eclipse

Pode-se então dizer que o Eclipse é uma *framework* multiplataforma, multilinguagem e multifuncionalidade, que oferece múltiplos *plugins* para a criação de *software*, tais como [14]:

- Editores de código, que permitem alterar o código fonte de qualquer aplicação a ser desenvolvida;
- Visualizar a hierarquia das classes em janelas;
- Visualizar os atributos e funções de uma classe em janelas;
- Criar listas de tarefas, para ajudar a fazer a gestão de trabalhos em equipa;
- Permite exportar a lista dos erros de compilação;
- Permite que o utilizador configure o ambiente gráfico do próprio Eclipse a gosto para um formato a que esteja mais familiarizado e que tenha as ferramentas que mais usa;
- Ferramentas que permitem integrar novos compiladores;
- Permite a depuração de código em tempo real.

2.3.3. .Net

O *.Net* é um ambiente de execução para gerir aplicações desenvolvidas para a *framework .Net*. Os serviços disponibilizados pela *framework .Net* para as aplicações em execução são os seguintes [15]:

- Gestão de memória. Em muitos programas os programadores são os responsáveis por fazer a alocação e libertação de memória e por gerir o tempo de vida dos objetos da aplicação. A *framework .Net* fornece estes serviços para a aplicação;
- Um sistema com tipos comuns. O *.Net* fornece um conjunto de tipos básicos que é comum a todas as linguagens alvo da framework do *.Net*;
- Ferramentas e tecnologias de desenvolvimento. A *framework .Net* disponibiliza várias bibliotecas nas mais diversas áreas de aplicação;

- Interoperabilidade de linguagens. Os compiladores que tenham como alvo uma plataforma *.Net* gerem um código intermédio com o nome de *Common Intermediate Language* (CIL), que é compilado em tempo de execução. Graças a isto, uma rotina escrita numa linguagem pode ser utilizada por outra linguagem completamente diferente;

- Execução lado a lado. A *framework* consegue resolver conflitos entre versões permitindo que múltiplas versões da *framework* coexistam no mesmo computador, possibilitando a existência de múltiplas aplicações desenvolvidas em versões diferentes;

- Compatibilidade entre versões. Com raras exceções, aplicações que tenham sido desenvolvidas em versões anteriores da *framework*, conseguem correr na *framework* mais atual.

Os objetivos desta *framework* são [16]:

- Fornecer um ambiente de programação orientada a objetos consistente, independentemente se a aplicação é para ser executada local ou remotamente;

- Fornecer um ambiente que minimize o conflito entre versões e o desenvolvimento de *software;*

- Fornecer um ambiente que promova a execução segura de código, incluindo o código criado por um autor desconhecido;

- Fornecer um ambiente que elimine os problemas de desempenho de ambientes interpretados;

- Torna a experiência do programador consistente entre vários tipos de aplicações diferentes;

- Construir todas as comunicações de acordo com os *standards* industriais, para garantir que todo o código feito para *.Net* seja compatível com qualquer outro código.

A figura 3 mostra a arquitetura do *.Net*, segundo [16] em grande detalhe.

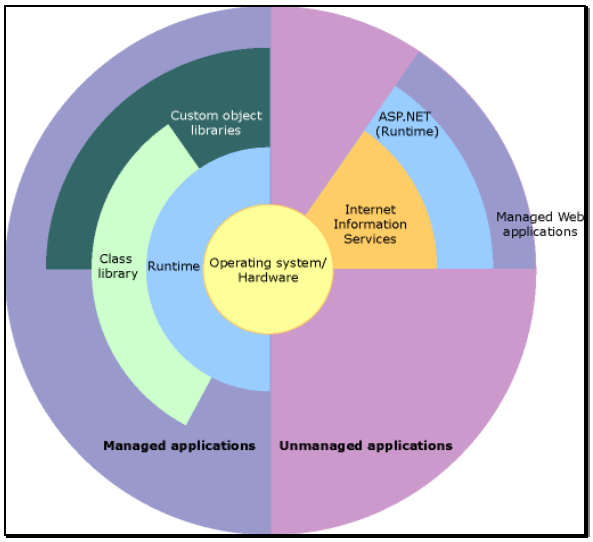


Figura 3 – Arquitetura do .Net

O Visual Studio é um IDE desenvolvido pela Microsoft que oferece suporte a uma larga gama de linguagens de programação. A última versão desta ferramenta, o Visual Studio 2012, apresenta uma maior quantidade de linguagens a que dá suporte, estendendo a plataforma para linguagens como o Java Script e o HTML5. Apesar de ser uma ferramenta proprietária, ou seja, é necessário pagar para se ter acesso a uma versão com todas as funcionalidades, a última versão, apresenta uma característica que elimina uma das principais limitações das antigas versões, ou seja, suporta multiplataforma, graças à nova *framework .Net*.

O Visual Studio é também um IDE que usa *plugins*, o que permite instalar mais ferramentas para desenvolver *software*, sendo uma delas o VMSDK.

2.3.3.1. VMSDK

Visualization and Modeling SDK [17] é um conjunto de ferramentas para o desenvolvimento de *software* que pode ser usado com o Microsoft Visual Studio. Este SDK era conhecido

como DSL Tool no Visual Studio 2005 mas o seu nome foi alterado para VMSDK nas últimas versões do Visual Studio.

O VMSDK permite criar uma DslDefenition (Domain-specific Language Definition) com classes, construindo o DSL classe por classe. Cada classe tem atributos que são propriedades do domínio. É também nas classes que se cria as relações entre elas para depois permitir a ligação entre componentes. Essas relações poderão ser:

- 0..*, um número arbitrário de vezes;
- 0..1, zero ou uma vez;
- 1..1, exatamente uma vez;
- 1..*, pelo menos uma vez.

Com as classes, o VMSDK cria uma linguagem visual constituída por formas. Vulgarmente utiliza caixas e setas, mas podem ser modificadas, alteradas ou melhoradas usando os decoradores fornecidos pelo VMSDK.

Depois das classes estarem criadas, das relações definidas e das formas visuais criadas, é possível lançar uma segunda instância, onde se pode arrastar e largar as formas de uma *toolbox* para uma área de edição, e conectá-las usando as relações criadas anteriormente para fazer o desenho do sistema desejado.

O VMSDK também permite fazer validações, parte muito importante para quem constrói um IDE para gerar código. Permite três tipos de validações [17]:

1. Por pedido, que permite que o desenho esteja errado mas quando uma condição de validação é ativada, como abrir, gravar e validar, o desenho é verificado e qualquer inconsistência no modelo é notificado ao utilizador;

2. Por regras intercetivas, que são regras que restringem a criação de ligações e/ou objetos errados;

3. E por regras fixas, que consiste em alterar automaticamente valores ou códigos gerados quando se altera algum aspeto do modelo.

Por fim permite gerar ficheiros de texto com qualquer tipo de extensão, usando *templates* de texto, que por sua vez conseguem gerar o código usando a DSL criada.

2.4. Conclusão

Tendo o autor analisado todos os fatores referidos anteriormente ao longo deste capítulo, o mesmo optou por utilizar o Visual Studio para desenvolver o IDE a que se propôs nesta dissertação. A escolha do autor por este IDE assenta, em primeiro lugar, no facto de ser um IDE com bastante uso, tanto na escola como na indústria atual, o que garante um maior número de documentação para ajudar na resolução de erros. Também por ser um IDE com a qual o autor já se encontra bastante familiarizado devido a trabalhos realizados nos anos anteriores.

Sendo o VMSDK uma ferramenta, que na opinião do autor, o ajudaria muito a criar o seu IDE, e sendo o mesmo um *plugin* do Visual Studio, foi um fator muito importante na escolha.

Com o VMSDK o autor poderá criar facilmente o seu próprio DSL, usar um editor especializado para definir o modelo do Sistema Operativo, criar automaticamente um editor gráfico para desenhar o sistema desejado, e gerar facilmente um código usando *templates* de texto que usam a DSL criada. Estes quatro fatores que a ferramenta oferece foram o que levou o autor a escolher o Visual Studio e o *plugin* VMSDK para fazer esta dissertação.

Todas as decisões tomadas pelo autor tiveram em conta a complexidade e tempo de que dispunha para fazer o projeto com o objetivo de cumprir o mesmo no tempo planeado. A utilização do Visual Studio permitiu reduzir o esforço do autor na adaptação da ferramenta, o VMSDK com as funcionalidades que oferece tornou possível agilizar a implementação.

Capítulo 3

STANDARDS

Este capítulo apresentará os *standards* necessários ao desenvolvimento da tese, a importância dos mesmos nesta dissertação e na área de aplicação. Será feita a exposição dos mesmos detalhadamente.

3.1. AUTOSAR

AUTOSAR (AUTomotive Open System ARchitecture) [18] define uma arquitetura *standard* e livre para o desenvolvimento de *software* automóvel, desenvolvido em conjunto com fabricantes de automóveis, fornecedores e os desenvolvedores de ferramentas. Esta arquitetura *standard* irá fornecer uma infraestrutura básica para ajudar no desenvolvimento de *software* para veículos, interfaces com utilizadores e a gestão de todos os domínios da aplicação. Tal inclui, como alguns dos principais objetivos, padronizar as funções básicas dos sistemas, permitir a fácil implementação em diferentes veículos e plataformas, integrar vários fornecedores de componentes, e facilitar a manutenção do produto durante todo o seu ciclo de vida e todas as atualizações do *software*.

O *standard* da AUTOSAR encontra-se dividido em várias camadas (Figura 4) para uma mais fácil interpretação e implementação:

1. A *Application Layer* é composta pelos diferentes componentes de *software* específicos de cada aplicação [18].

2. Na camada do *AUTOSAR Runtime Environment* encontram-se os serviços de comunicação. Esta camada é responsável por permitir que a comunicação seja uniforme, pois a interface para comunicação inter e intra ECU é igual [18].

3. A camada do *Basic Software* é responsável por abstrair a aplicação do *hardware*. Esta camada está ela própria dividida em vários níveis de abstração. Na camada de *Basic Software* pode ainda ser incluído um sistema operativo [18].

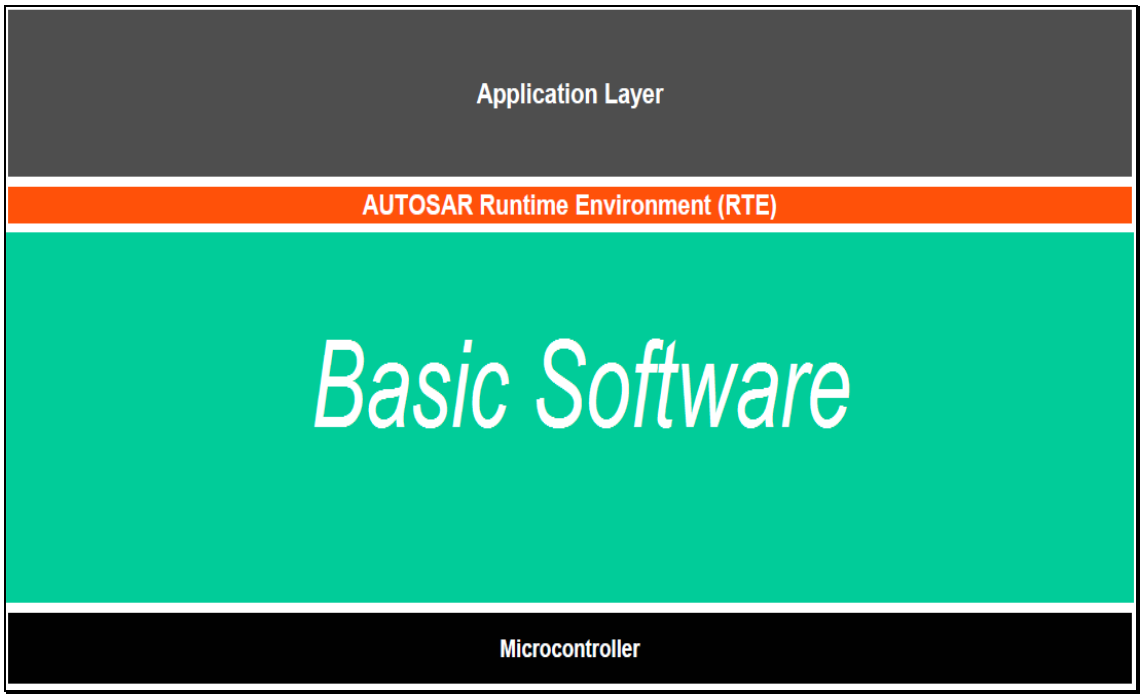


Figura 4 – Pilha de *Software* AUTOSAR: Divisão das camadas[19]

Os sistemas operativos definidos pelo *standard* da AUTOSAR obedecem às especificações OSEK (*Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen*; em inglês: "*Open Systems and their Interfaces for the Electronics in Motor Vehicles*") [2]. O OSEK é um *standard* industrial que estabelece as especificações para o RTOS usado nas unidades de controlo de veículos. Define uma interface padrão para sistemas operativos para facilitar a portabilidade de aplicações de *software e* aumentar a interoperabilidade, a reutilização e a troca de pacotes de *software* independentemente do *hardware* utilizado [20].

3.1.1. *Basic Software*

A camada do *basic software* divide-se em várias camadas (Figura 5) que por sua vez se subdividem em outras camadas:

1. A *Services Layer* é a camada superior do *basic software* e é responsável por fornecer às camadas superiores da pilha AUTOSAR (ver figura 4) as funcionalidades do SO. É ainda responsável pela gestão de serviços e comunicações na rede do veículo, pelos serviços de diagnósticos, pelo estado do ECU, e por conseguir que as *layers* superiores sejam independentes do ECU.

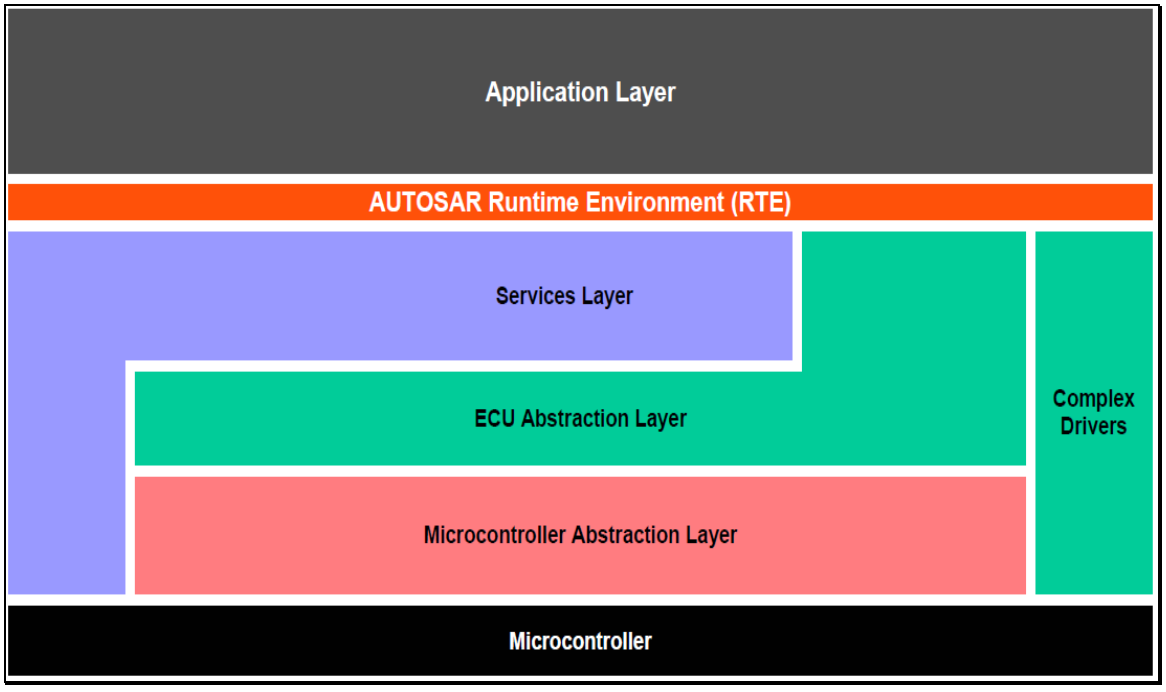


Figura 5 – Pilha de *Software* AUTOSAR: Subcamadas da camada *Basic Software*[19]

2. ECU *Abstraction Layer*, camada responsável por facultar às camadas superiores a interface com os *drivers* no *Microcontroller Abstraction Layer* e fornecer os *drivers* para

os dispositivos externos, tem como função facultar APIs uniformes para o acesso aos dispositivos independentemente das respetivas localizações.

3. A camada do *Microcontroller Abstraction Layer* fornece os *drivers* para os dispositivos internos e permite que a implementação se torne independente do *hardware* ao ter uma interface uniforme com as camadas superiores e independente do microcontrolador.

4. A camada *Complex Drivers* fornece uma *fastpath* para dar resposta às necessidades de atuadores e sensores críticos que precisam de acesso direto ao microcontrolador, evitando desde modo possíveis *overheads* devido à intervenção do RTOS, para cumprirem requisitos funcionais e temporais.

3.1.1.1. *Services Layer*

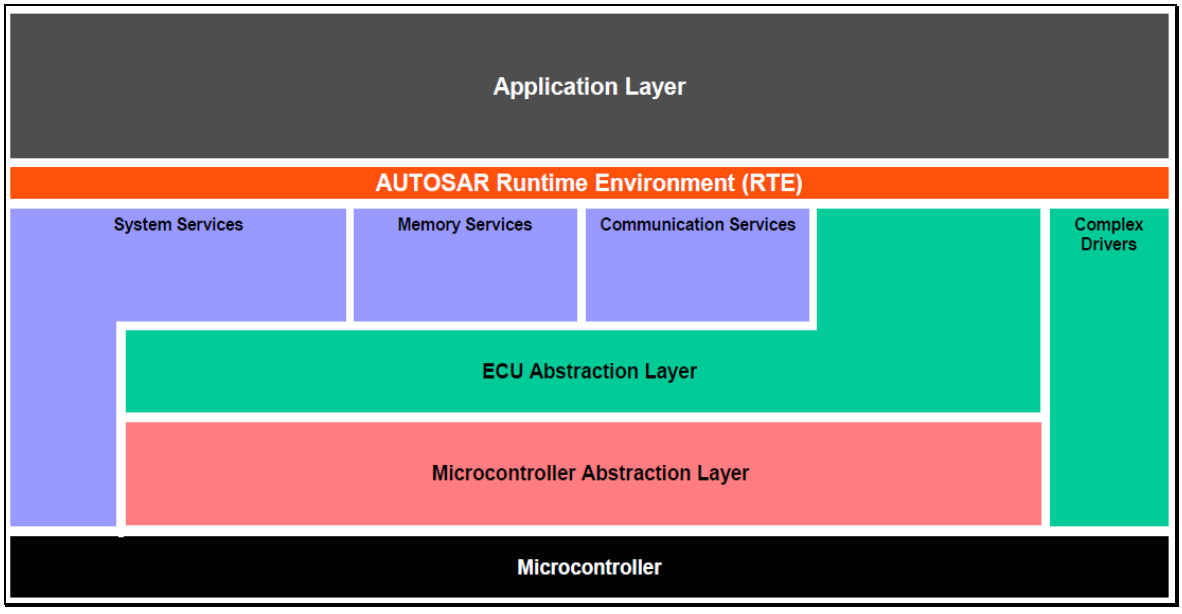


Figura 6 – Pilha de *Software* AUTOSAR: Subcamadas da subcamada *Services Layer*[19]

A camada *Services Layer* encontra-se subdividida em três camadas (Figura 6) responsáveis por fornecer APIs à aplicação:

1. Na camada de *communication services* encontram-se as APIs responsáveis por facultar ao utilizador uma interface uniforme com a rede do veículo, APIs de diagnóstico e APIs de gestão da rede. Para assim esconder os diferentes protocolos e propriedades das mensagens da aplicação.

2. Na camada *memory services* estão as APIs responsáveis pelo acesso uniforme aos dados não voláteis, por exemplo, *save, load, chechsum,* abstraindo assim a aplicação das propriedades e localizações das diferentes memórias.

3. A camada *system services* é a única que comunica com todas as outras, pois é responsável por fornecer os serviços básicos, como é o caso do acesso do sistema operativo ao *timer*, e partes que sejam dependentes da aplicação e do *hardware*.

3.1.1.2. ECU *Abstraction Layer*

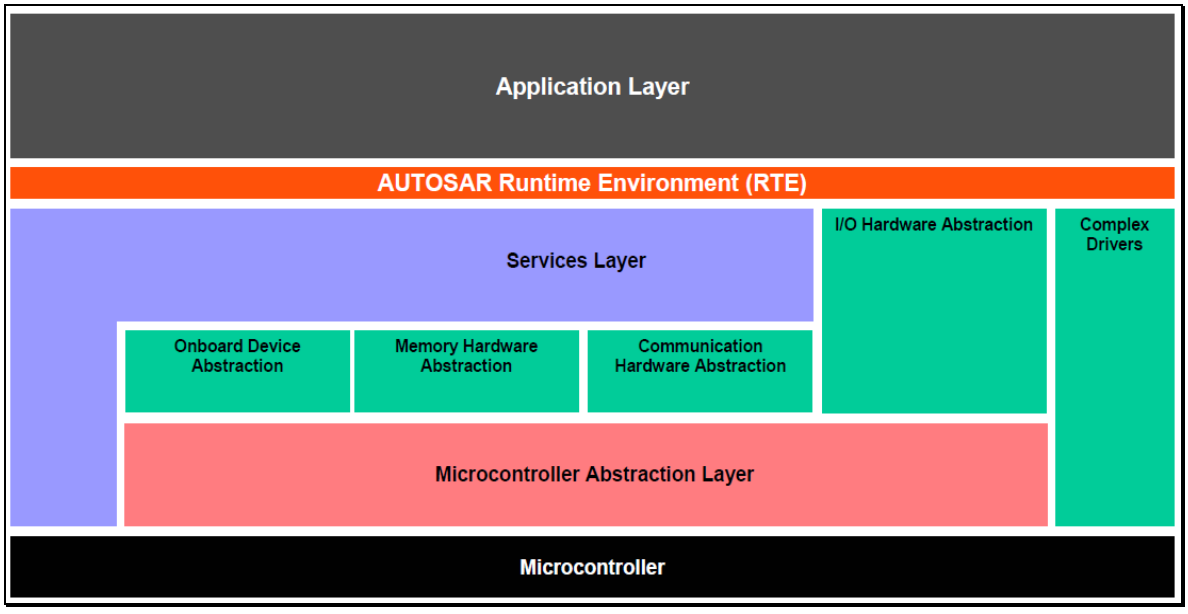


Figura 7 Pilha de *Software* AUTOSAR: Subcamadas da subcamada *Abstraction Layer*[19]

A camada *ECU Abstraction Layer* encontra-se subdividida em quatro camadas (Figura 7): a camada de *I/O Hardware Abstraction*, a de *Communication Hardware Abstraction*, a de

Memory Hardware Abstraction e a de *Onboard Device Abstraction*. Estas quatro camadas são responsáveis por definir o acesso uniforme aos *drivers*. Tornando assim a aplicação independente do *hardware*.

Na camada *I/O Hardware Abstraction*, as propriedades do *hardware* e *layout* são escondidas das camadas superiores, os dispositivos de I/O são acedidos através de um *signal*, e os *signals* são interpretados como estando diretamente ligados ao *hardware*.

A camada *Communication Hardware Abstraction* permite abstrair da localização dos controladores de comunicação e do *hardware layout* dos mesmos, pois fornece os mesmos mecanismos de acesso aos barramentos, independentemente da localização, *On-Chip* ou *On-Board*.

A camada *Memory Hardware Abstraction* permite abstrair da localização e do tipo de memória a ser utilizada por fornecer mecanismos uniformes para o acesso a diferentes tipos de memórias e os mesmos mecanismos de acesso para memórias internas e externas.

A camada *Onboard Device Abstraction* é responsável por definir o acesso aos *drivers* para os periféricos específicos de cada ECU, como o exemplo *timers* e *whatchdog*.

3.1.1.3. Microcontroller Abstraction Layer

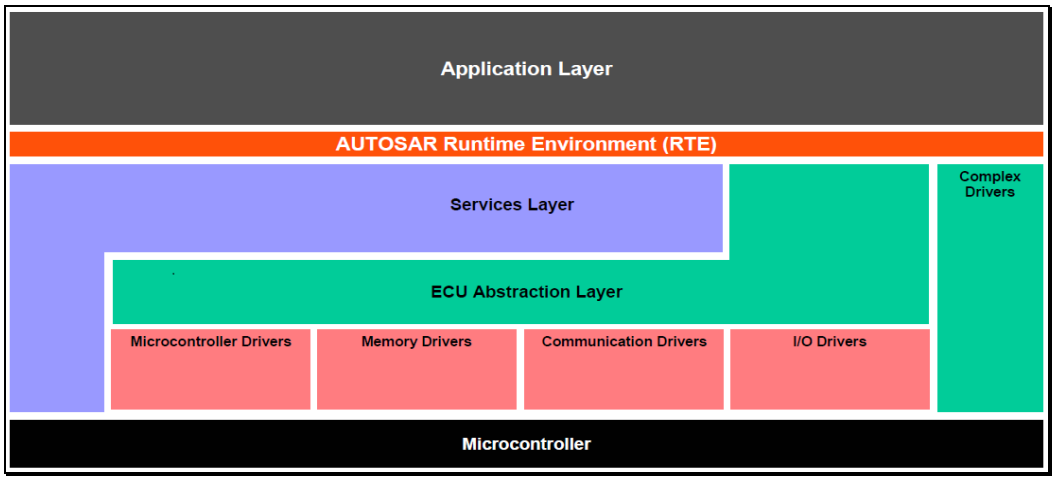


Figura 8 – Pilha de *Software* AUTOSAR: Subcamadas da subcamada *Microcontroller Abstraction Layer*[19]

A camada *Microcontroller Abstraction Layer* encontra-se subdividida em quatro camadas (Figura 8): a camada de *I/O Drivers*, a de *Communication Drivers*, a de *Memory Drivers* e a de *Microcontroller Drivers*, sendo estas quatro camadas as responsáveis por fornecer os *drivers* que permitem que a aplicação interaja com o *hardware*.

Na camada *I/O Drivers* estão os *drivers* para os dispositivos I/O, como sensores e atuadores mais simples e os periféricos *On-Board*.

A camada *Communication Drivers* fornece os *drivers* para os sistemas de comunicação *On-Board* e as redes de comunicação do veículo.

Por sua vez, a camada *Memory Drivers* fornece os *drivers* para o acesso às memórias internas e externas.

Na camada *Microcontroller Drivers* encontram-se os *drivers* dos periféricos específicos do ECU.

3.2. OSEK/VDX

Em Maio de 1993, o OSEK [21] foi iniciado num projeto alemão de um conjunto de várias empresas da indústria automóvel, com o objetivo de criar um *standard* que especificasse a arquitetura, o interface e o comportamento dos Sistemas Operativos para sistemas embebidos usados em veículos.

Os parceiros iniciais no projeto foram a BMW, Bosh, DaimlerChrysler, Opel, Siemens, VW e com o IIIT da Universidade de Karlsruhe como coordenador do projeto. Em 1994 juntaram-se os fabricantes de carros franceses PSA e Renault, introduzindo no projeto o seu VDX, que é um projeto semelhante ao OSEK mas entre a indústria francesa. Sendo por fim em Outubro de 1995 que o grupo apresenta o OSEK/VDX como resultado final da integração dos dois *standards*.

O *standard* é formado por três áreas. A primeira área é de comunicação, de dados trocados internamente na unidade ou entre unidades. A segunda área é do sistema operativo, a execução em tempo real de *software* no ECU. E por fim, a área da gestão de redes, configuração e monotorização da rede.

Os principais objetivos do *standard* é dar suporte à portabilidade de código e reutilização do mesmo, através de: especificação de interfaces o mais abstratas possíveis e aplicações o mais independentes possíveis. O *standard* especifica a interface para o utilizador que é independente da rede e do *hardware*, e um projeto eficiente da arquitetura. O mesmo permite ajustes opcionais na arquitetura para melhor se ajustar à aplicação.

O *standard*, para além de definir todos os aspetos do sistema operativo, também definiu uma nova linguagem própria, OIL (OSEK Implementation Language). Como todos os objetos são definidos durante o projeto da aplicação, o OIL foi criado como uma linguagem simples, para definir os atributos desses objetos.

3.2.1. OIL

O OIL permite definir as propriedades dos objetos a ser usados, e quando compilado cria as estruturas necessárias e ficheiros necessários em código C. A aplicação, que é composta pelo código de cada tarefa, é escrito em C e compilado pelo compilador de C. O código C resultante da compilação do OIL é também compilado e fica assim com os ficheiros do tipo objeto. O código do sistema operativo passa pelo mesmo processo, sendo que no fim o *linker* transforma todos os ficheiros objetos num ficheiro executável. Todo este processo é ilustrado pela Figura 9.

Repare-se que o *linker* apenas fundirá na geração do executável (Figura 9) os ficheiros objetos necessários para a aplicação conforme especificado no ficheiro OIL, excluindo todos os restantes componentes e contribuindo deste modo para a redução do *footprint* de memória.

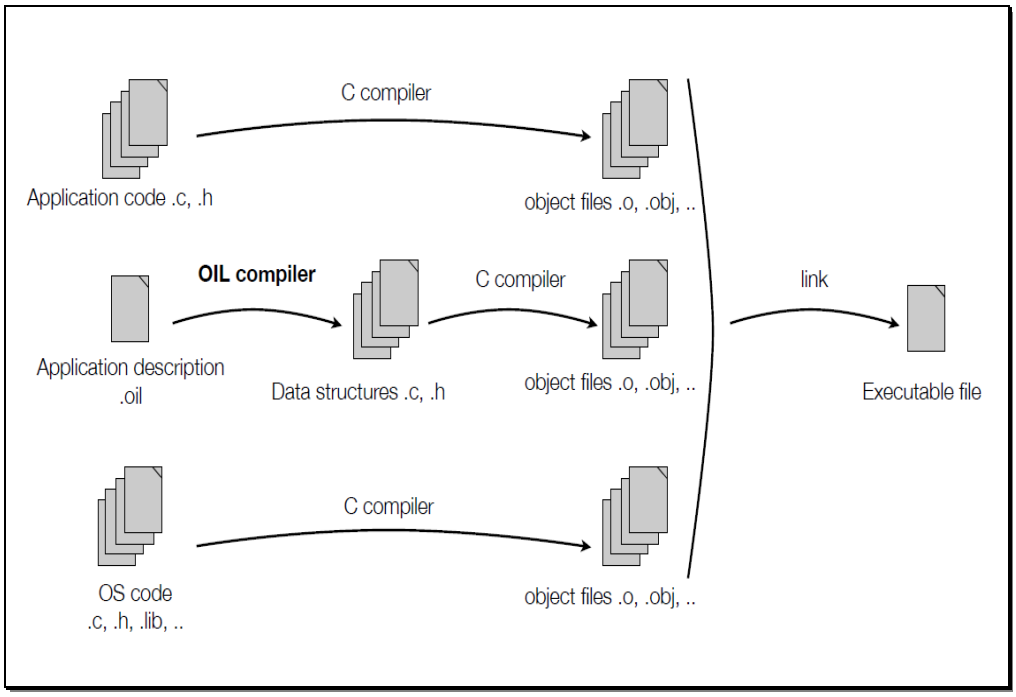


Figura 9 – Esquema de compilação OSEK

3.2.2. **OSEK OS** [22]

O OSEK OS define as caraterísticas que um sistema operativo para sistemas embebidos deve ter para poder ser compatível com o AUTOSAR. Essas características especificam que: (1) tem de existir uma fácil portabilidade de aplicações entre arquiteturas de sistemas operativos OSEK, (2) que a configuração da aplicação é estática, isto é, a arquitetura da aplicação é totalmente conhecida no momento da compilação, (3) que permita incorporar apenas as funções do SO que são realmente usadas, e (4) que tenha um comportamento previsível, comportamento esse que é um requisito de aplicações em *real-time*.

3.2.2.1. Sumário

O OSEK OS proporciona vários serviços. Esses serviços estão divididos em grupos e estruturados da seguinte forma:

- Gestão de Tarefas. Permite ativar e desativar Tarefas, e também fazer a gestão do estado das Tarefa e trocar a Tarefa que está ativa;

- Sincronização. O sistema operativo tem de suportar dois tipos de sincronismo para Tarefa, o "*Resource management*", gestão de operações que não possam ser interrompidas, como o acesso a recursos ou dispositivos; e também "*Event control*", que é a gestão de eventos para manter o sincronismo das Tarefas;

- Gestão de Interrupções. Fornece serviços de gestão a interrupções de tarefas;

- Alarmes. Dois tipos de alarme: alarmes relativos e absolutos;

- Tratamento de mensagens entre tarefas. Tem serviços para permitir a troca de dados entre tarefas;

- Tratamento de Erros. Mecanismos de suporte para o utilizador em caso de erro, para vários tipos de erros diferentes;

3.2.2.2. Tarefa

O OSEK define dois tipos diferentes de Tarefas: as Básicas e as Estendidas. Os dois tipos de Tarefa são muito parecidos, com apenas um estado a mais no caso das Estendidas. As Tarefa não possuem qualquer parâmetro, por isso todo e qualquer dado e informação tem de ser passado por mensagens. Uma Tarefa pode ser ativada mais que uma vez, a que se dá o nome de "*Multiple activation requests*", e essa Tarefa será executada o número de vezes que for ativada. Se tiver suspensa, o valor de ativação será registado e a Tarefa será ativada múltiplas vezes depois, desde que não exceda o valor limite definido para essa Tarefa.

Uma tarefa pode-se auto-terminar, ou seja, só a própria tarefa é que pode terminar a si mesma, mas nenhuma outra pode terminá-la. Contudo, pode sempre ser ativada por outra tarefa.

As Tarefas Básicas têm três estados (Figura 10): "*running*", que é quando a Tarefa se encontra em execução, "*ready*", quando uma Tarefa se encontra pronta para ser executada

para estar à espera que o processador seja libertado, e o último estado "*suspended*", que é quando uma Tarefa se encontra suspensa, esperando que haja um sinal de ativação.

Uma "Tarefa Básica" só liberta o processador em três situações muito destintas: quando a mesma já terminou a sua execução e assim cede o processador a outra, quando uma Tarefa mais prioritária é ativada e então a comutação de contexto é efetuada e o processador é libertado para a Tarefa mais prioritária, ou então quando ocorre uma interrupção, em que a comutação de contexto também é efetuada.

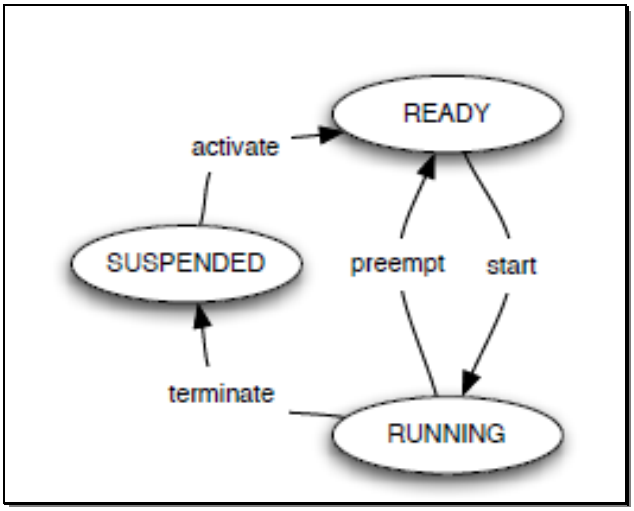


Figura 10 – Estados possíveis para uma Tarefa Básica

As Tarefas Estendida têm os mesmos três estados das "Tarefas Básicas", o "*running*", o "*ready*", e o "*suspended*", mais um quarto "*waiting*" (Figura 11), que é um estado intermédio da Tarefa, quando está à espera de um Evento para poder continuar a sua execução.

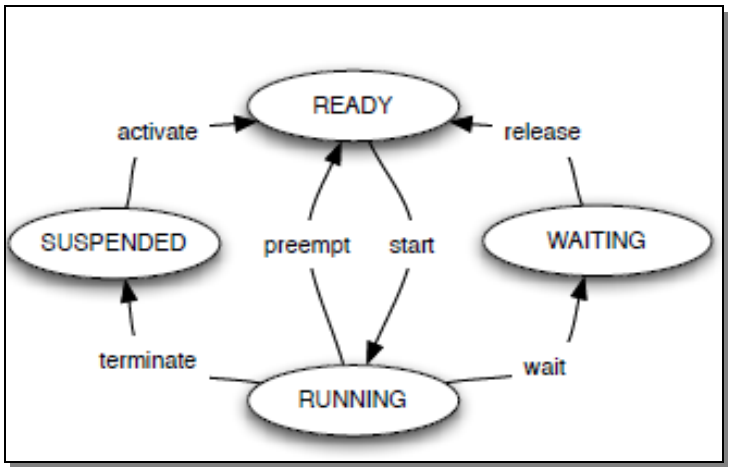


Figura 11 – Estados possíveis para uma Tarefa Estendida

E uma Tarefa Estendida só liberta o processador quando, tal como a Tarefa Básica, termina a sua execução, uma tarefa mais prioritária é ativada ou uma interrupção acontece, mas ao contrário da Tarefa Básica também liberta o processador quando é feita uma chamada ao sistema do tipo "*WaitEvent*"

3.2.2.3. Política de escalonamento

O tipo de escalonador usado pelo sistema pode ser de três tipos diferentes:

- Full preemptive;
- Non preemptive;
- Mixed preemptive.

O tipo de escalonador depende do que for definido em cada Tarefa. Se todas as Tarefas forem definidas com um escalonador "*Full*", então sempre que uma Tarefa de maior prioridade é ativada uma comutação de contexto será desencadeada. Se for "*Non"*, só é chamado o escalonador e provocada a comutação de contexto quando a Tarefa acaba a sua execução, e é feita uma chamada explícita do escalonador ao sistema, ou a Tarefa é colocada no estado "*Waiting*". O último tipo é o "*Mixed*", que consiste na fusão das duas estratégias anteriores de

escalonamento. Conforme a Tarefa que esteja em execução num determinado momento, o escalonador assume o tipo definido por essa Tarefa.

3.2.2.4. Interrupções

O *standard* define dois tipos de interrupções: as ISR categoria 1 e 2. As de categoria 1 não podem usar serviços do sistema operativo, as APIs. No fim da ISR categoria 1, a execução retorna exatamente ao mesmo ponto de onde tinha parado. Não tem qualquer influência na gestão das Tarefas, gera um menor *overhead*, e precisa de menos tempo para fazer a comutação de contexto. Em contrapartida, as IRS categoria 2 podem usar serviços do sistema operativo (APIs), mas antes de poderem ser executadas têm que garantir que nenhum recurso ou evento se encontra bloqueado.

3.2.2.5. Eventos

Os eventos são mecanismos usados para assegurar o sincronismo. Só Tarefas Estendidas podem ter eventos associados. Podem estar associados a uma Tarefa um ou mais eventos, e o mesmo pode estar associado a mais que uma Tarefa. Qualquer Tarefa pode ativar o evento de outra mas apenas a Tarefa a que ele pertence pode limpar o evento ou ficar a esperar que seja ativado. Assim uma Tarefa Estendida pode ficar à espera de alguma ação que outra faça e ser posteriormente notificada por meio de eventos.

3.2.2.6. Gestão de Recursos

A gestão de recursos tem conjuntos de regras e restrições que são cumpridos para evitar que *Deadlocks* ocorram devido ao uso de recursos. Os recursos têm também associados uma prioridade, e quando uma tarefa ou uma interrupção acede a um recurso, a mesma herdará

momentaneamente a prioridade do recurso, se a prioridade deste for maior que a prioridade da tarefa ou interrupção. Duas tarefas ou interrupções não podem aceder simultaneamente ao mesmo recurso, e o acesso a recursos nunca coloca ama tarefa no estado "*waiting*".

Uma restrição é que enquanto um recurso estiver ocupado não pode ser chamada nenhuma das seguintes funções:

- TerminateTarefa;
- ChainTarefa;
- Schedule;
- WaitEvent.

Quando uma tarefa ou interrupção ocupa múltiplos recursos devem ser libertados seguindo uma filosofia LIFO (Last In First Out), ou seja, devem ser libertados pela ordem inversa pela qual foram acedidos.

O escalonador também pode ser visto como um recurso, um recurso que pode ser acedido por todas as Tarefas. Tem como nome predefinido RES_SCHEDULER e é criado automaticamente, e mesmo que uma Tarefa bloqueie o escalonador das restantes tarefas, as interrupções continuam a ser respondidas.

A prioridade aos recursos é atribuída estaticamente no momento de geração do sistema, para evitar problemas de inversão de prioridade e *deadlocks*. O cálculo das prioridades dos recursos tem de cumprir o seguinte conjunto de regras para o cálculo das prioridades:

- Prioridade maior ou igual que a da tarefa/interrupção de maior prioridade que usa o recurso, e menor que a prioridade de tarefas/interrupções que não usam o recurso mas têm maior prioridade que as que usa. Como se pode ver no exemplo da Figura 12.

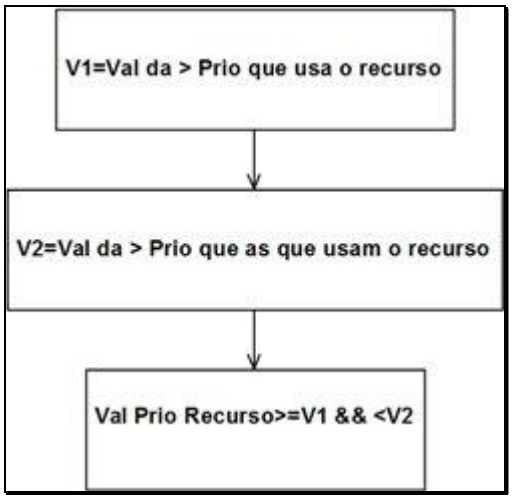


Figura 12 – Protocolo de atribuição da Prioridade dos recursos

3.2.2.7. Alarmes

O *standard* OSEK OS define configurações diferentes para os alarmes: o "Alarme Relativo" que usa um valor relativo em relação ao do contador, ou o "Alarme Absoluto" que usa o valor do contador. Tem também o "Alarme Unico" que apenas ocorre uma vez e é desativado, e tem de ser reativado pela Tarefa de interrupção de atendimento ao alarme, ou então o "Alarme Cíclico" que não precisa de reativação e permite um controlo mais preciso do período do mesmo.

Os alarmes usam contadores para saberem os tempos de ativação, mas o sistema não fornece APIs *standard* para a manipulação direta de contadores. O sistema operativo é responsável pelas ações de gestão dos contadores que acionam os alarmes. Os sistemas operativos OSEK oferecem sempre pelo menos um contador, que pode ser derivado de HW ou SW.

Um alarme pode estar diretamente ligado ou associado a um *alarm-callback routine*, quando ISRs de categoria 2 estão desativadas, ou seja, um alarme que ative outro alarme. As *alarm-callback routines* não possuem parâmetros nem valor de retorno. Um alarme apenas pode estar associado a um contador ou à ativação de uma Tarefa.

3.2.2.8. Mensagens

Os sistemas operativos OSEL definem que um sistema de mensagens tem sempre que existir, isto é tem de haver mecanismos para comunicação entre tarefas executadas no processador. De acordo com o OSEK/COM, o sistema tem no mínimo de implementar CCCA (*Commonication Conformance Class Internal Comunication*) para assegurar as comunicações internas do processador entre tarefas.

3.2.2.9. Tratamento de Erros

Para o tratamento de erros, os sistemas operativos OSEK têm de fornecer rotinas específicas, denominadas "*Hook routines*", que são associadas a outras funções que podem ser executadas antes ou depois da invocação destas. Permitem ver o estado do sistema pois são parte do sistema operativo, e nunca são interrompidas pois têm prioridade maior que qualquer tarefa. Com essas rotinas é possível desligar o sistema, reiniciar, ou desencadear qualquer outra ação que o programador ache adequada quando ocorre um erro.

3.2.3. RTOS OSEK-ENABLED

Diz-se que um RTOS é OSEK-ENABLED, se cumprir todos os requisitos que foram abordados no ponto anterior. Para esta dissertação o autor escolheu o RTOS Trampoline, por se tratar de um RTOS que cumpre os requisitos OSEK/VDX e tem as mesmas APIs que são especificadas pelo OSEK/VDX, mas ainda se encontra à espera de certificação [23].

3.3. IP-XACT

IP-XACT é um *standard* desenvolvido pela SPIRIT Consortium [24] como uma nova forma de permitir a configuração, integração automática e reutilização de IPs. É um *standard* que utiliza o formato XML para descrever os componentes eletrónicos e os seus projetos, independentemente da linguagem de implementação do componente [25]. Tal significa que o *standard* não disponibiliza qualquer informação sobre a implementação do componente, apenas a informação relativa ao interface do mesmo com o sistema, os compiladores usados, o mapeamento da memória e a localização de ficheiros externos ao *standard*, ficheiros onde poderão estar, ou não, a implementação ou informações sobre a implementação do componente.

Os objetivos principais do *standard* passam por: assegurar a compatibilidade entre vários componentes de múltiplos vendedores diferentes, facilitar a troca de bibliotecas complexas de componentes entre EDA (*electronic design automation*) e ferramentas para o desenho de SoC (*System-on-Chip*), e descrever componentes configuráveis através de metadata [26].

Este *standard* foi aprovado com o número 1685 pelo IEEE a 9 de Dezembro de 2009, e foi publicado a 18 de Fevereiro de 2010.

O *standard* não tem nenhuma extensão oficial para IPs de Software [26], pelo que nesta dissertação o autor decidiu criar as suas próprias extensões, que poderão ser facilmente alteradas para as oficiais, um dia que as mesmas sejam publicadas.

Nesta dissertação foi implementada uma *framework* orientada a modelos e compatível com AUTOSAR para gerar e gerir IPs individuais e o próprio RTOS, que posteriormente serão referenciados como blocos básicos da *framework* ou ambiente de desenvolvimento AUTOSAR. Uma das primeiras etapas consistiu no *refactoring* do trampoline OS de acordo com um modelo IP-XACT estendido e posterior integração do IP do OS no repositório IP-XACT.

Refactoring é o processo de modificar a estrutura de um programa sem modificar o comportamento do mesmo [27]. Quando se faz o *refactoring* de um *software* procura-se melhorar um ou mais de quatro aspetos [28]:

- Qualidade;
- Reutilização;
- Manutenção;
- Extensibilidade.

O *refactoring* que vai ser feito ao *software* nesta dissertação forcar-se-á principalmente na reutilização do código. Será desenvolvida uma *framework* que permite reutilizar código, gerando código já testado automaticamente, mas sem alterar o funcionamento já esperado do *software*.

Capítulo 4

ESPECIFICAÇÃO DO SISTEMA

Neste capítulo será apresentada a análise e *design* do sistema, com o intuito de escolher a melhor implementação possível. O resultado deste trabalho servirá como base à construção do trabalho final. Em primeiro lugar será abordado neste capítulo todas as funcionalidades do sistema e restrições previstas para o mesmo. Depois será apresentada a modelação feita para a estruturação do sistema operativo no IDE. Por fim será descrito como se vai proceder à geração do código.

4.1. Trampoline OS

Cada vez mais nos dias que correm, um sistema operativo capaz de correr em microcontroladores é uma peça fundamental no desenvolvimento de aplicações. Permite abstrair os programadores da complexidade do *hardware*, fornece mecanismos para uma melhor proteção da memória, facilita a gestão das tarefas e permite o controlo de periféricos do microcontrolador de uma forma mais simples e sem precisar de grandes conhecimentos do *hardware*. O sistema operativo serve então como uma ligação entre o *hardware* e o utilizador ou programador do sistema, proporcionando um ambiente simples e mais seguro para a execução de programas.

Como o Trampoline é um sistema operativo orientado a objetos, traz algumas vantagens relativamente aos tradicionais sistemas operativos [29] oferecendo uma programação imperativa. Como a programação orientada a objetos permite herança, encapsulamento, recursividade ou até polimorfismo, garante que será mais simples a gestão da complexidade assim como a manutenção do código. Como a utilização de classes torna o código mais

modular e de fácil leitura para qualquer outro programador, aumenta também assim a possibilidade de reutilização de código, que é exatamente o que se espera nesta dissertação: a criação de um IDE capaz de gerar o código para a aplicação desejada pelo utilizador, utilizando código já existente.

Para a construção do IDE é necessária uma análise do sistema operativo e dos seus componentes. Depois de analisar o sistema operativo e cada componente do mesmo, obteve-se um diagrama de classes UML (Figura 13) que servirá como base à construção do IDE.

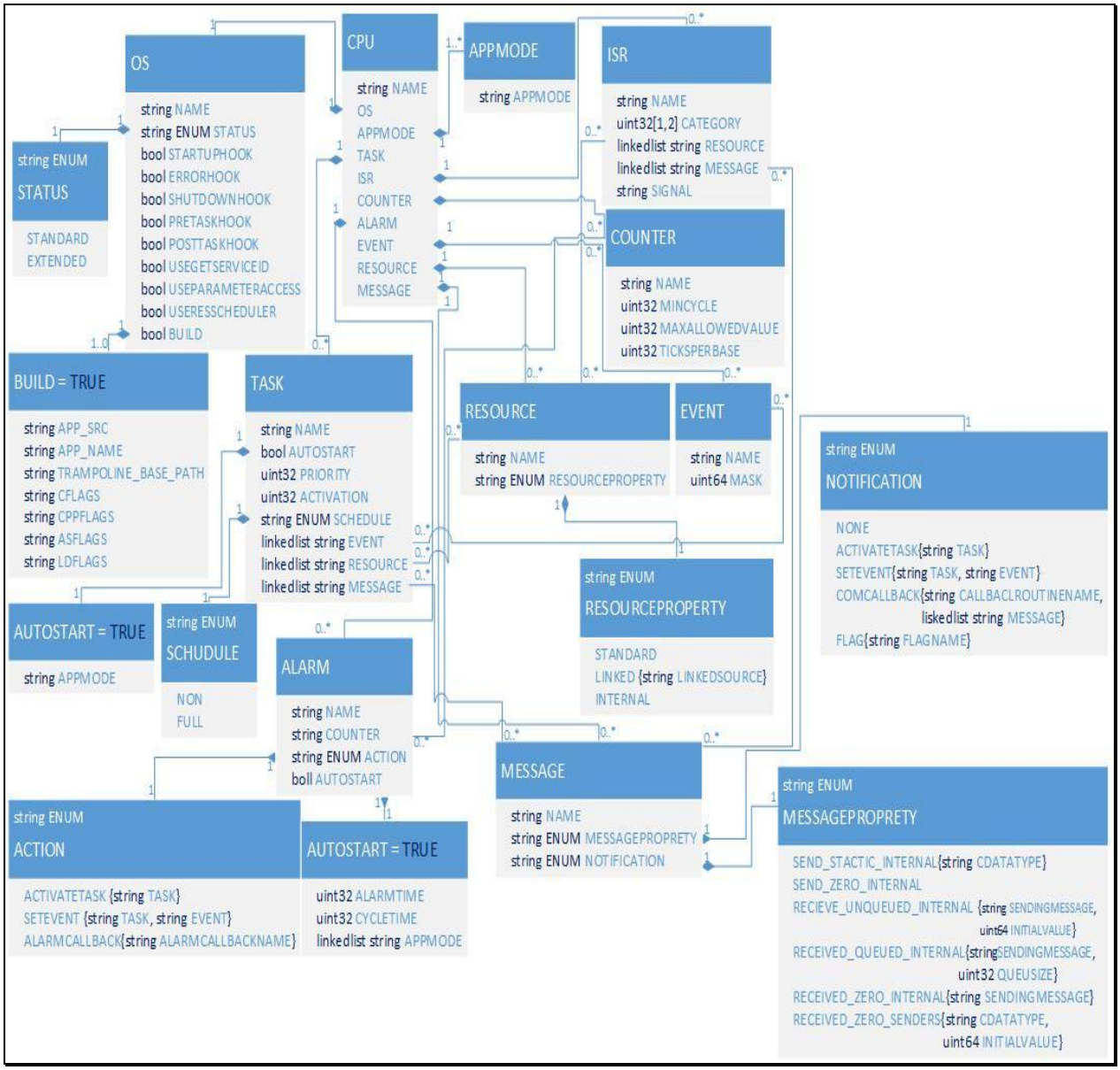


Figura 13 – Arquitetura estática do Sistema Operativo representada por diagrama de classes

Usando o UML é possível prever logo alguns componentes que são sempre obrigatórios quando se desenha o sistema. Esses componentes [30] são o CPU, que é o componente base de todo o sistema, o OS, que configura as propriedades da aplicação, e o APPMODE, que define o modo da aplicação.

Os restantes componentes serão ou não incorporados na aplicação dependendo do que o utilizador precise de utilizar [30]: o componente TASK, que permite que o sistema tenha zero ou mais tarefas, o componente RESOURCE, que é usado para definir a lista de tarefas que o consome, o EVENT, que permite definir uma lista de eventos com as quais algumas tarefas, as do tipo *extended*, poderão interagir, o COUNTER, que permite definir um contador que serve como base dos mecanismos dos alarmes definido pelo componente ALARM, o ISR, que representa as rotinas de serviço a interrupções, e o componente MESSAGE, que dá acesso ao sistema de mensagens do sistema operativo.

Tendo esta análise como base é então possível começar a planear o trabalho que será necessário efetuar para obter o resultado esperado, assim como a abordagem a esse trabalho, e também a análise a todas as funcionalidades e restrições impostas por esta mesma.

4.2. Abordagem

É importante delinear logo de início as fases de construção de um IDE e a abordagem a seguir, para que assim se evite ter de voltar várias vezes atrás para corrigir ou acrescentar novas funcionalidades ou regras. Com isto aumenta-se a produtividade e reduz-se o tempo que será necessário para a implementação do IDE e todas as suas funcionalidades [31].

O IDE gráfico terá duas instâncias a serem implementadas. A primeira será a do programador do IDE, enquanto que a segunda será a que o utilizador irá usar. Na primeira instância será feito um diagrama, usando o UML do sistema operativo, para representar o mesmo e todas as suas diferentes partes, assim como todas as ligações, e ainda definir as regras de validação do sistema operativo. Por isso a primeira instância será chamada de gerador do IDE. Na segunda

instância será feito o desenho do sistema operativo desejado, e serão verificadas as regras escritas na primeira instância. Esta segunda instância será também responsável por gerar o código final. Por seu lado, a segunda instância será também dividida em duas partes. A primeira parte será a do IDE gráfico, que permite o desenho do sistema operativo, e a segunda parte terá o gerador de código que irá gerar o código C, OIL e XML.

4.2.1. Primeira Instância

A primeira instância, que permite depois gerar o IDE gráfico, será construída a partir de várias classes. Cada componente do sistema operativo será representado por uma classe, e nessa classe estarão todas as instanciações necessárias para o componente. Cada classe terá os construtores específicos de cada campo para cada componente, assim como os assessores aos mesmos e os assessores das ligações entre componentes. Essas classes terão também todas as validações necessárias.

As validações são criadas especificamente para cada componente, pois cada componente tem requisitos únicos e regras únicas que deve respeitar. Para facilitar ao utilizador a interpretação dos erros e a sua correção, surgiu a necessidade de criar mensagens específicas para cada validação e de as apresentar ao utilizador no momento em que cria um componente, para que saiba do que esse componente precisa, e também quando for para gerar os ficheiros de código, pois algumas validações só podem ser efetuadas no momento final antes da geração do código.

Para que se possa criar e apresentar ao utilizador essas mensagens é necessário, no Visual Studio, criar recursos para isso. Esses recursos consistem na criação de mensagens predefinidas a serem chamadas nas validações, quando necessárias. Os recursos criados permitem que as mensagens sejam exibidas no painel de erros do Visual Studio, permitindo assim dar ao projeto um aspeto mais limpo e profissional.

Alguns componentes podem ter ligações entre eles, uns podem estar ligados a vários do mesmo tipo e outros apenas a um. Para representar essas possíveis ligações serão usadas duas formas diferentes, uma para ligar um componente a vários do mesmo tipo e outra para ligar

um componente apenas a um de outro tipo. As ligações de um componente a vários serão representadas por setas, enquanto que as ligações de um para um serão descritas no próprio componente como mais um elemento das suas propriedades.

As ligações de um para um são fáceis de criar e gerir pois é preciso apenas acrescentar um campo às propriedades de um componente. As ligações de um para muitos implicam a criação de setas, que são objetos a ser desenhados, e como tal será preciso criar os construtores desses objetos e a imagem dos mesmos. Requer também uma maior complexidade na descrição das regras, para que as ligações sejam corretas, assim como uma maior complexidade na geração de código, visto que estes tipos de ligações são depois mais difíceis de aceder e logo de codificar o que elas representam.

Todos os componentes terão também de ser desenhados e terá de ser também possível editar as suas propriedades. Para tal é preciso criar os seus objetos e imagens. Cada componente terá uma representação própria com propriedades predefinidas mas que serão editáveis e sejam independentes entre componentes do mesmo elemento, ou seja, dois componentes TASK têm as mesmas propriedades mas cada um deles pode ter valores diferentes nas propriedades.

Para que todos os objetos criados estejam de fácil acesso no momento do desenho, quer sejam componentes ou ligações, é necessária então a criação de uma *toolbox* que esteja acessível no momento de desenho. Esta *toolbox* será chamada de Trampoline pois dentro dela estarão todos os objetos que representam os componentes do sistema operativo, bem como as suas ligações. Assim, no momento do desenho do sistema operativo por parte do utilizador, tudo o que o mesmo precisará estará disponível num único sítio de fácil acesso e uso, tornando assim a primeira instância uma base para criação do IDE.

4.2.2. Segunda instância

A segunda instância será onde o utilizador irá trabalhar, isto é, onde projetará o sistema operativo, usando todos os recursos construídos na primeira instância para desenhar e validar o sistema operativo. Será também nesta segunda instância que, usando recursos pertencentes à mesma, será possível gerar todos os ficheiros de código propostos neste projeto.

O utilizador começará por selecionar e arrastar os blocos de componentes que precisa a partir *toolbox* para a área de desenho, fazendo assim o primeiro esboço do sistema operativo desejado. Para que o sistema seja completo ele deve ainda usar as setas de ligação disponíveis na *toolbox* para representar ligações entre componentes ou então, no caso das ligações de um para um, alterar o campo correspondente a essa ligação no próprio componente para assinalar a mesma. Por fim, para que o sistema seja exatamente aquilo que o utilizador pretende, ele deve ainda adaptar adequadamente as propriedades de cada componente que desenhou e assim gerar uma versão do sistema operativo com todas as funcionalidades pretendidas.

Depois de o desenho estar feito é esperado que sejam gerados três ficheiros, cada ficheiro contendo o necessário para que seja possível compilar o sistema operativo com as especificações esperadas pelo utilizador. Os ficheiros são um com o código C da aplicação, outro com o código OIL da configuração do sistema, e por fim um terceiro ficheiro XML com o IP-XACT. Com apenas os dois primeiros será possível compilar o sistema operativo pronto a executar. O ficheiro com a descrição, segundo o *standard* IP-XACT do sistema operativo, servirá para uma melhor integração do sistema operativo criado em outros ambientes de desenvolvimento, permitindo uma melhor interceção com IDEs previstos para desenvolver tanto *software* como *hardware*, além de permitir que seja futuramente criado um IDE que interprete unicamente o XML e recrie o sistema operativo, facilitando assim a partilha de aplicações.

4.3. IP-XACT

O ficheiro IP-XACT criado pelo IDE contém três partes muito importantes: a primeira parte com a informação do componente, a segunda parte com a localização de ficheiros adicionais e a terceira parte com as extensões do vendedor, extensões que serão criadas pelo autor para descrever o *software*, visto que o *standard* ainda não tem nenhuma extensão oficial.

4.3.1. Componente

A primeira parte do ficheiro XML é a parte "*component*", é onde se encontra a informação básica do componente, como é visível na Figura 14.

```
<spirit:component>
    <spirit:vendor>Trampoline</spirit:vendor>
    <spirit:library>Trampoline</spirit:library>
    <spirit:name>tp1</spirit:name>
    <spirit:version>1.0</spirit:version>
</spirit:component>
```

Figura 14 – Exemplo de um componente IP-XACT: Campo "*component*"

O vendedor e biblioteca são sempre os mesmos por defeito, pois trata-se de um sistema operativo que pertence a uma marca. O nome é dado pelo utilizador, para que com cada versão do sistema operativo que deseje criar tenha um nome identificável, e a versão do mesmo para permitir que na biblioteca vá guardado diferentes sistemas com o mesmo nome mas com configurações diferentes.

4.3.2. *FileSet*

O segundo bloco é o dos "*fileSets*", secção onde se encontra a localização de ficheiros adicionais ao componente. Mas, nesta dissertação, apenas os ficheiros que não são ficheiros IP-XACT estarão no "*fileSets*". No caso desta dissertação, todos os ficheiros descritos nesta secção do ficheiro são os necessários para que o sistema possa ser compilado.

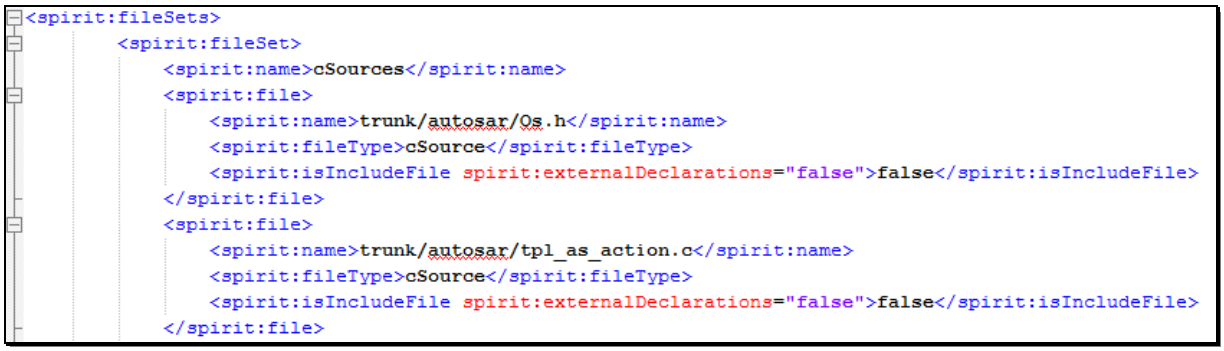


Figura 15 – Exemplo de um componente IP-XACT: Campo "*FileSets*"

Como é possível ver na Figura 15, o "*fileSet*" dos ficheiros está dividido em tipos de ficheiro, depois dentro de cada tipo tem os vários ficheiros, tendo o seu nome, que é também onde se encontra a sua localização, o seu tipo de ficheiro, no caso da imagem dois ficheiros de código C, e também se se trata de um ficheiro IP-XACT ou não.

4.3.3. Extensões

Na última parte do ficheiro XML é onde se encontra os "*vendorExtensions*". É nessa secção que o fornecedor ou desenvolvedor do componente pode adicionar as suas próprias extensões para descrever o componente, quando as que já estão definidas no *standard* não são suficientes.

Para esta dissertação, o autor também terá de criar extensões adicionais, pois o *standard* não tem nenhuma extensão oficial para *software*.

As novas extensões criadas serão para permitir ter no ficheiro XML uma descrição pormenorizada da aplicação criada. Isto permitirá também uma mais fácil implementação de algumas propostas de trabalho futuro descritas no capítulo 7.

```
<spirit:vendorExtensions>
    <esrg:OS>
        <OS:Name>config1</OS:Name>
        <OS:STATUS>EXTENDED</OS:STATUS>
        <OS:STARTUPHOOK>False</OS:STARTUPHOOK>
        <OS:ERRORHOOK>False</OS:ERRORHOOK>
        <OS:SHUTDOWNHOOK>False</OS:SHUTDOWNHOOK>
        <OS:PRETASKHOOK>False</OS:PRETASKHOOK>
        <OS:POSTTASKHOOK>False</OS:POSTTASKHOOK>
        <OS:USEGETSERVICEID>False</OS:USEGETSERVICEID>
        <OS:USEPARAMETERACCESS>False</OS:USEPARAMETERACCESS>
        <OS:USERESSCHEDULER>False</OS:USERESSCHEDULER>
        <OS:BUILD>True</OS:BUILD>
    </esrg:OS>
    <esrg:build>
        <OS:APP_SRC>lab.c</OS:APP_SRC>
        <OS:TRAMPOLINE_BASE_PATH>../..</OS:TRAMPOLINE_BASE_PATH>
        <OS:APP_NAME>elab</OS:APP_NAME>
        <OS:CFLAGS>-Wall</OS:CFLAGS>
        <OS:CPPFLAGS></OS:CPPFLAGS>
        <OS:ASFLAGS></OS:ASFLAGS>
        <OS:LDFLAGS></OS:LDFLAGS>
    </esrg:build>
```

Figura 16 – Exemplo de um componente IP-XACT: Campo "*VendorExtensions*"

Como é possível ver na Figura 16, as extensões que serão criadas têm como inspiração a própria especificação do sistema operativo, para que quando houver extensões oficiais seja mais fácil de compreender o ficheiro e assim fazer a alteração para as oficiais.

4.4. Funcionalidades e Restrições

Como o desenvolvimento de aplicações *bare-metal* implica um elevado conhecimento da arquitetura interna do microcontrolador e que a programação seja feita em baixo nível, aumenta a probabilidade de se cometer erros ou obter alguns resultados indesejados, que podem ser dispendiosos a nível de tempo e custos para corrigir. Assim, o uso de um RTOS simplifica a construção das aplicações de *software*, tornando o sistema mais seguro e fiável.

O *standard* IP-XACT foi desenvolvido para componentes de *hardware*, permitindo apenas a associação do módulo *software* a executar. Contudo permite a criação de extensões que poderão ser adicionadas para representação isolada de IPs de *software* como desejado neste trabalho. A ideia consiste em também representar um componente IP-XACT de *software*, como o sistema operativo, e posteriormente associar o mesmo a um IP de *hardware* que o irá executar. Assim sendo, a plataforma de desenvolvimento terá que ser capaz de verificar e garantir que tal IP de *software* apenas será instanciado na área de desenho caso esteja associado a um IP de *hardware*.

Como já foi descrito nos objetivos, o projeto proposto para esta dissertação passa por desenvolver um IDE que permita desenhar e gerar rápida e facilmente o código de um sistema operativo.

Um IDE permite que a geração de todos os códigos propostos nesta dissertação sejam assim mais robustos pois evita que sejam cometidos erros por parte do programador, tornado não só mais rápida a criação de um novo código, mas também fazendo com que seja necessário despender menos tempo em testes e correção de *bugs*, pois o código base de cada componente gerado pelo IDE já terá sido testado e corrigido.

É por isso importante analisar as funcionalidades que o IDE precisa de oferecer e as restrições que lhe estão implícitas.

A primeira funcionalidade é permitir desenhar o sistema operativo, ou seja, criar uma representação do sistema operativo desejado, sem que para isso o utilizador precise de fazer um estudo ou uma análise intensiva da documentação do mesmo. Utilizando essa representação feita pelo utilizador, é esperado que o IDE produza o código OIL e C do sistema operativo, pronto a ser compilado. Também se espera que seja criado um ficheiro XML com a descrição IP-XACT específica para o sistema operativo desenhado.

É esperado que o IDE, durante o desenho do sistema operativo e no momento de geração dos ficheiros de código, faça validações do que o utilizador está a desenhar e apresente mensagens de erro, caso os haja. As validações a efetuar são verificações de intervalos de números, nomes e ligações válidas, e se os mínimos são selecionados e configurados, deixando que as verificações finais sejam feitas depois pelos compiladores de C e OIL.

4.4.1. Gerador do IDE

No gerador do IDE será desenhado o diagrama de classes UML do sistema operativo, e com esse desenho serão criados todos os componentes que o sistema operativo oferece, não sendo necessário depois o utilizador utilizar todos. Cada componente terá associado a si um construtor, que permite desenhar o mesmo componente, e poderá ter conceções de e para outros componentes para descrever ligações entre componentes.

Cada componente tem um conjunto específico de campos a ser preenchido com as configurações desejadas para o sistema operativo. Assim, cada componente terá um conjunto de regras e validações específicas dependendo dos campos que tem e da própria funcionalidade do componente.

Analisando cada componente separadamente, é possível descrever quais as validações internas de cada um e as suas regras específicas.

Como também é necessário criar ligações entre os componentes, é preciso ver quais as ligações válidas e que tipo de ligações usar.

4.4.1.1. Validação e Restrições dos Componentes

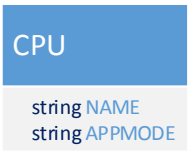


Figura 17 – Componente: CPU

Para o componente CPU (Figura 17), um componente que não é desenhado, visto que é a base do desenho e todos os outros serão desenhados dentro deste, não é necessário validar a sua presença, visto que ele já lá está e é impossível de apagar. É no entanto necessário verificar se o utilizador escreve duas *strings*, que não sejam nulas e sejam válidas, sendo que por uma

questão de simplificação, e para que no momento inicial da criação não haja tantas mensagens de erros, o campo NAME e APPMODE já tem valores predefinidos, que são passiveis de alteração.

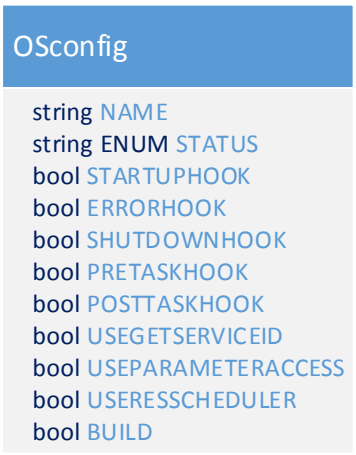


Figura 18 – Componente: OSconfig

O componente OSconfig (Figura 18) é um dos componentes obrigatórios para que o sistema operativo possa ser gerado e mais tarde compilado. Por isso é apresentada uma mensagem de erro caso este componente não esteja desenhado. O campo NAME é verificado para que não seja nulo, nem uma *string* inválida, tendo um valor já predefinido de possível edição. STATS tem duas opões possíveis:

- EXTENDED;
- STANDARD.

Sendo assim, o utilizador só pode selecionar uma das duas opções e assim evita-se erros. Os campos do tipo booleano também não precisam de verificações pois só têm as opções "*True*" ou "*False*". O campo BUILD define se o componente CBUILD é necessário ou não, sendo feita a regra de verificação se o componente CBUILD está presente quando BUILD é "*True*".

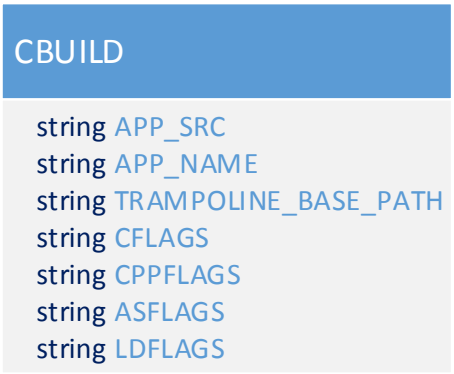


Figura 19 – Componente: BUILD

Além de se verificar se CBUILD (Figura 19) está presente quando o campo BUILD do componente OSconfig está a "*True*", também é feita a verificação de todas as *strings* para que não sejam nulas e sejam válidas.

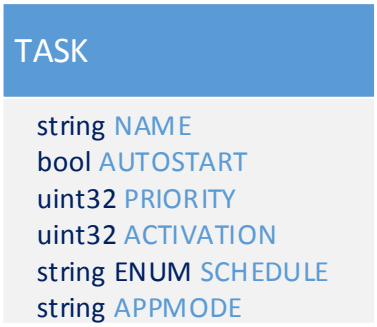


Figura 20 – Componente: TASK

No componente TASK (Figura 20) já existem mais validações que os componentes anteriores. Em primeiro lugar é verificado o campo NAME, que tem de ser uma *string* válida e não nula, além de única, pois podem existir várias TASKs no sistema operativo mas todas com nomes diferentes. Se AUTOSTART for "*True*", é verificado se APPMODE é válido, senão não é necessário pois não será usado. PRIORITY e ACTIVATION têm os seus valores validados para serem maiores que zero e com máximo de valor que seja possível de conter num uint32. O SCHEDULE só aceitará um de dois valores à disposição do utilizador, que são:

- FULL;
- NON.

Figura 21 – Componente: ISR

O componente ISR (Figura 21) necessitará de validações básicas. NAME e SIGNAL serão verificados como duas *strings* se são válidas e não nulas, além de que cada NAME é único. Por fim CATEGORY apenas aceitará um de dois valores à disposição do utilizador, 1 ou 2.

Figura 22 – Componente: EVENT

EVENT (Figura 22) apenas terá verificado o campo NAME, para ver se é uma *string* válida, não nula, e não repetida noutros EVENTs.

Figura 23 – Componente: RESOURCE

No componente RESOURCE (Figura 23) é verificado se NAME é uma *string* válida, não nula, e não repetida noutros RESOURCEs. Para o campo RESOURCEPROPERTY é dado ao utilizador três opções para que escolha uma:

- INERNAL;
- LINKED;
- STANDARD.

Figura 24 – Componente: COUNTER

COUNTER (Figura 24) terá de ter verificações para que MINCYCLE, MAXALLOWEDVALUE e TICKSPERBASE sejam maiores que zero e com um valor máximo que seja possível de conter num *uint32*. Além de que NAME terá de ser uma *string* válida, não nula e diferente entre todos os componentes do tipo COUNTER.

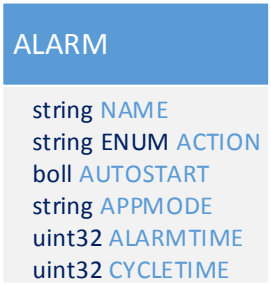


Figura 25 – Componente: ALARM

NAME terá de ser uma *string* válida, não nula e diferente entre todos os componentes do tipo ALARM (Figura 25). Para o campo ACTION, o utilizador poderá escolher uma de três possibilidades:

- ACTIVATETASK;
- ALARMCALLBACK;
- SETEVENT.

AUTOSTART é do tipo booleano aceitando apenas "*True*" ou "*False*", e se for definido como "*True*" é preciso verificar APPMODE, ALARMTIME e CYCLETIME, senão são dispensáveis essas verificações. Para APPMODE é verificado se é uma *string* válida e não nula, e para ALARMTIME e CYCLETIME é necessário verificar que sejam maiores que zero e com máximo de valor que seja possível de conter num *uint32*.

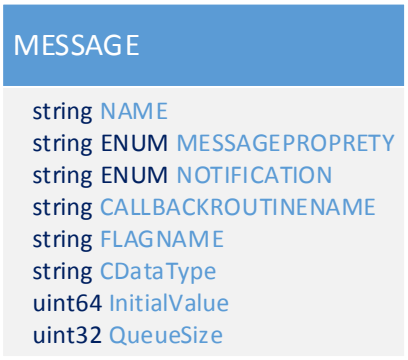


Figura 26 – Componente: MESSAGE

MESSAGE (Figura 26) é um componente complexo com validações com várias dependências. NAME terá de ser uma *string* válida, não nula e diferente entre todos os componentes do tipo ALARM. MESSAGEPROPERTY e NOTIFICATION têm várias opções para configuração, da qual o utilizador escolhe uma. Serão as seguintes:

- MESSAGEPROPRETY:

- RECEIVE_QUEUED_INTERNAL;
- RECEIVE_UNQUEUED_INTERNAL;
- RECEIVE_ZERO_INTERNAL;
- RECEIVE_ZERO_SENDERS;
- SEND_STATIC_INTERNAL;
- SEND_ZERO_INTERNAL.

- NOTIFICATION:
  - ACTIVATETASK;
  - COMCALLBACK;
  - FLAG;
  - NONE;
  - SETEVENT.

Dependendo das opções escolhidas nos dois anteriores, pode ser preciso verificar ou não os campos seguintes, pois cada campo tem utilidade para alguma das opções e não para todas. Assim, em caso de necessidade de validação, as *strings* são verificadas como válidas e não nulas, enquanto que os números como sendo maiores que zero e possíveis de conter em *uint64* e *uint32*, dependendo da configuração.

4.4.1.2. Ligações entre componentes

O componente CPU engloba todos os componentes, por isso pode-se dizer que todos eles são desenhados dentro de CPU, o que significa que todos estão implicitamente ligados ao CPU, tornando assim desnecessária a representação dessas ligações.

Os componentes OSconfig e CBUILD apenas se ligam um ao outro quando necessário, e como é uma ligação de um para um, visto que apenas um componente de cada um pode ser desenhado, a sua ligação é representada por uma seta, disponível na *toolbox* para que o utilizador represente a ligação.

O componente TASK pode estar ligado a vários componentes, com ligações em que uma TASK pode ter vários componentes do mesmo tipo. Esses componentes são EVENTs, RESOURCEs e MESSAGEs. Para essas ligações é criada uma seta que vai de TASK ao componente. Para cada tipo de ligação, ou seja, cada ligação de tipo diferente, sendo que ligar TASK a RESOUCE é um tipo e TASK a EVENT é outro, será criada uma seta na *toolbox*, que pode ser usada repetidamente para ligações do mesmo tipo, e uma seta diferente para cada tipo.

O componente ALARM está também ligado a vários componentes, mas neste caso cada alarme liga apenas a um componente de cada tipo. Assim, como ALARM liga apenas a uma TASK, um EVENT e a um outro ALARM, este outro ALARM é para casos em que é preciso que um ALAM ative outro. Assim, as ligações são representadas nas propriedades do ALARM, em que o mesmo tem um campo para cada componente a que se liga e uma lista dos componentes disponíveis desse tipo. Assim apenas é necessário escolher o nome do componente e a ligação é efetuada.

O componente ISR também pode estar ligado a vários componentes, numa ligação de um para muitos. Como ISR pode estar ligado a vários RESOURCEs e várias MESSAGEs, estas representações serão também representadas por uma seta, tal como acontece para TASK.

RESOURCEs podem estar associados a outros RESOURCEs. Como é uma ligação de um para um, a ligação é feita com a adição de um campo extra nas propriedades de RESOURCE para que se possa escolher um RESOURCE da lista dos que estão desenhados.

MESSAGE apenas se liga de um para um a TASK, EVENT e MESSAGE, por isso essas ligações serão representadas com a adição de três campos extra nas propriedades, uma para cada componente, com a lista dos componentes do seu tipo desenhadas até ao momento, para que o utilizador possa escolher.

4.4.2. Gerador de Código

O gerador de código será composto por três painéis principais: a *toolbox*, a Área de desenho e as Propriedades.

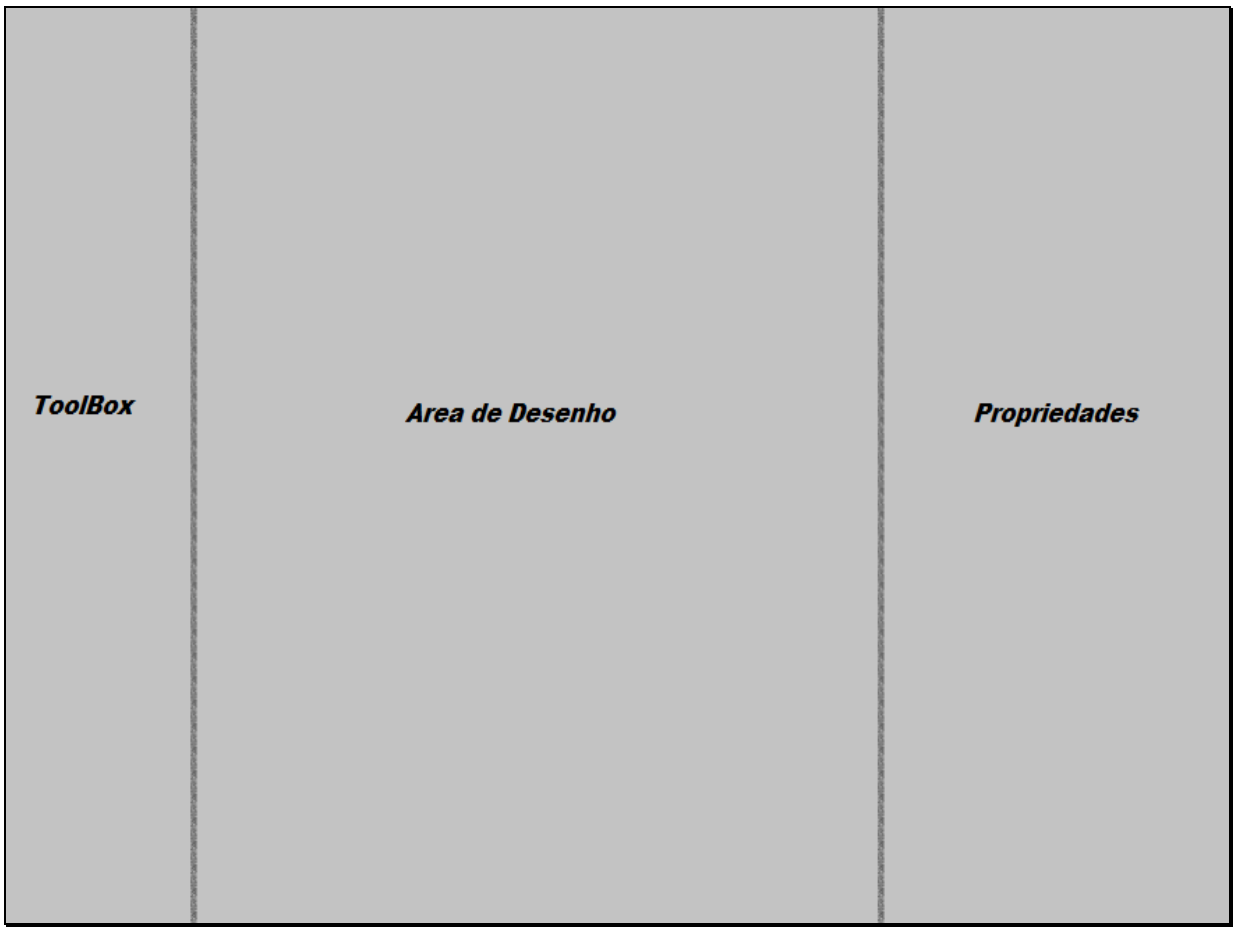


Figura 27 – Esquema do Interface do Gerador de Código

Na t*oolbox* estarão disponíveis todos os componentes e setas de ligação para serem arrastados para a Área de Desenho. Para todos os componentes que estiverem na Área de Desenho serão gerados os códigos necessários. Selecionando um componente na Área de Desenho são apresentadas as suas propriedades no painel Propriedades, sendo também possível editar as mesmas.

Tendo o utilizador desenhado todo o seu sistema operativo, é preciso gerar os ficheiros de código que representam o mesmo. Para cada sistema operativo em concreto serão gerados três ficheiros: um primeiro com o código C que terá o código executável da aplicação; um ficheiro com código GOIL que terá todas as configurações do sistema operativo para a sua execução, é também o primeiro a ser compilado pelo compilador GOIL, permitindo criar todo o código necessário para depois compilar o executável final com um compilador para código C, junto com o ficheiro C já criado; o terceiro ficheiro é um ficheiro XML com a descrição IP-XACT do sistema operativo.

Capítulo 5

IMPLEMENTAÇÃO

Neste capítulo será descrita toda a implementação do IDE, cumprindo as funcionalidades e restrições descritas no capítulo anterior. Será apresentada cada fase da construção do IDE, começando pela primeira instância com o gerador do IDE gráfico, a criação de componentes, e as regras de verificação e validação dos mesmos. Depois na segunda instância será explicado como se faz o desenho do sistema operativo e como serão gerados os diferentes ficheiros de código a partir do desenho.

5.1. Instanciação do IDE

Para gerar o IDE gráfico, a segunda instância, é necessário primeiro fazer a modelação do sistema operativo na primeira instância. Com o sistema todo modelado é preciso proceder à criação de todos os componentes e ligações que foram descritos no capítulo 4. É também preciso configurar todos os campos que cada componente tem e os seus valores por defeito, além dos campos extra para ligações como explicado anteriormente. Por fim, e ainda na primeira instância, será monstrado o processo da criação de regras para a validação.

5.1.1. Modelação do sistema para criação do DSL

Para criar o DSL que dá suporte a criação de uma semântica gráfica para o desenho do sistema operativo, com o uso de um editor grafico, e facilitar a geração de código de forma

generativa usando *templates* para os artefactos, é preciso modelar o sistema na ferramenta que o *.Net* proporciona.

Para a modelação do sistema é usado o diagrama UML anteriormente desenvolvido e, a partir do mesmo, é desenhada a modelação do sistema operativo (Figura 28), com todos os seus possíveis componentes e campos, como é visível nas seguintes figuras:

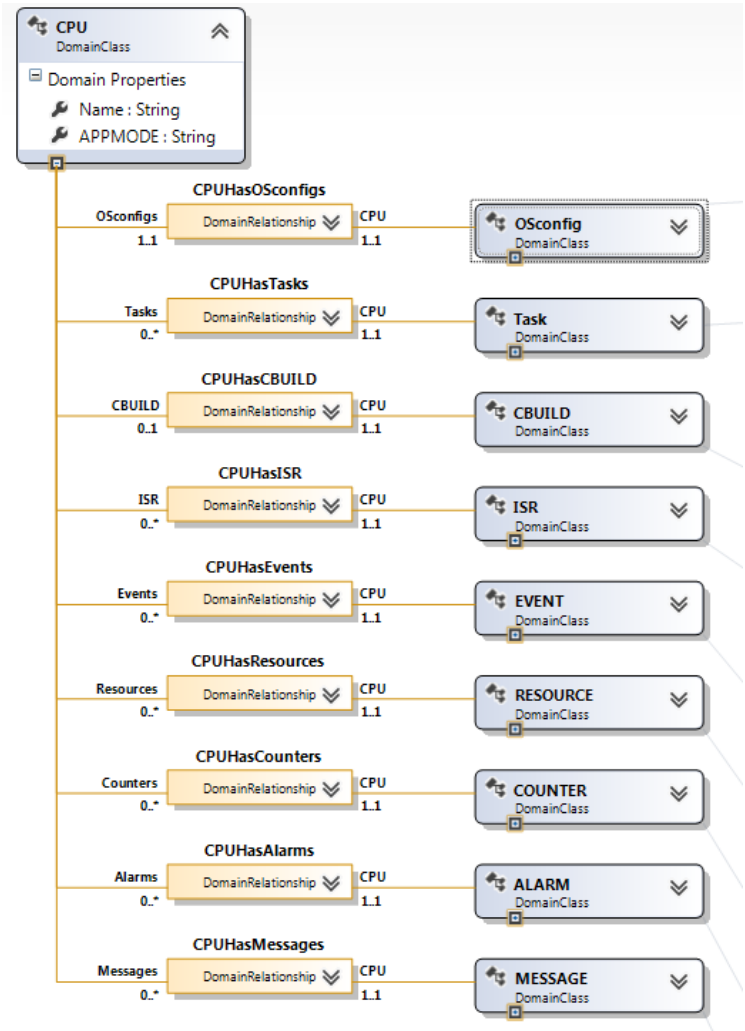


Figura 28 – 1.1.1. Modelação do sistema para criação do DSL: Modelo Geral do OS

Como foi explicado no capítulo 4, o CPU é a base do sistema, e por isso todos os outros componentes estão ligado ao CPU, e como tal não é necessário representar essas ligações no desenho fisicamente, já que todos os componentes serão simplesmente desenhados dentro do

bloco gigante que é CPU. As figuras seguintes monstram cada componente com os seus campos e possíveis ligações.

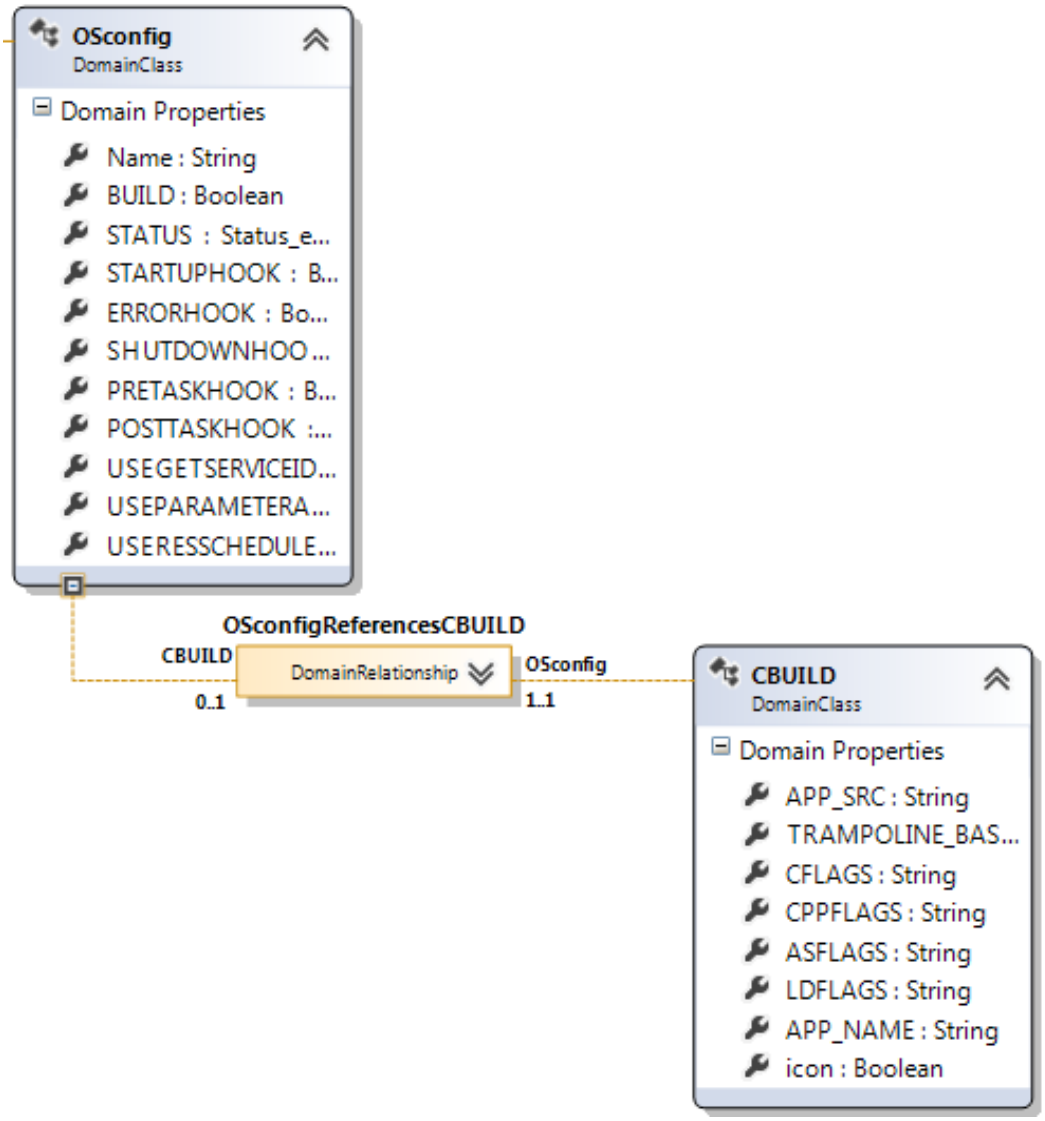


Figura 29 – Modelação do sistema para criação do DSL: OSconfig

OSconfig (Figura 29) está implementado com todos os seus campos e a sua única possível ligação que é a CBUILD. Essa ligação só é necessária quando BUILD é "*True*". Todas as validações e verificações serão explicadas no tópico seguinte.

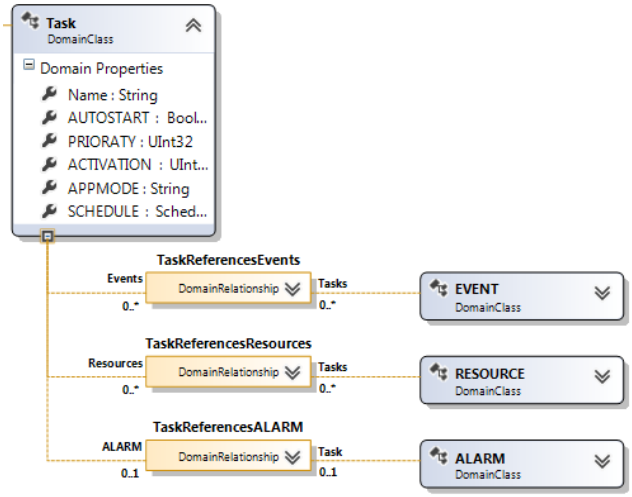


Figura 30 – Modelação do sistema para criação do DSL: TASK

TASK (Figura 30) já é implementado com várias possíveis ligações, e todos os seus campos de configuração.

Todos os outros componentes são implementados de forma similar e sempre obedecendo às especificações impostas no capítulo 4. As figuras 31, 32, 33, 34 e 35 mostram todos os componentes implementados.

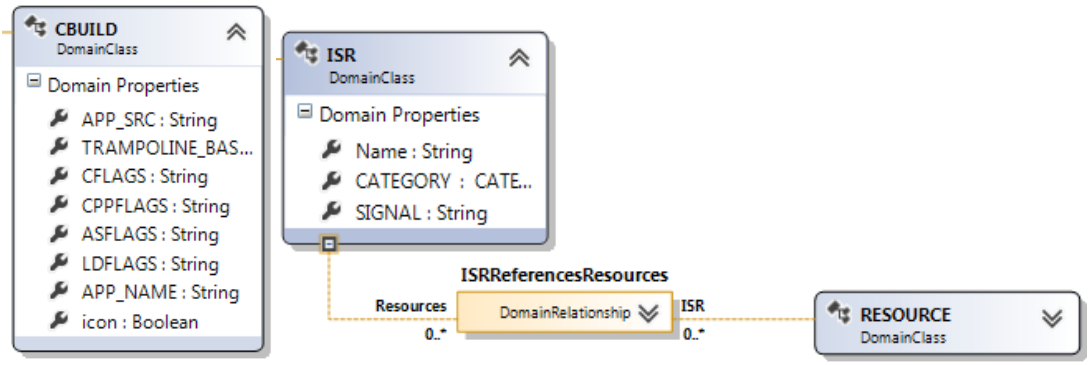


Figura 31 – Modelação do sistema para criação do DSL:: CBUILD e ISR

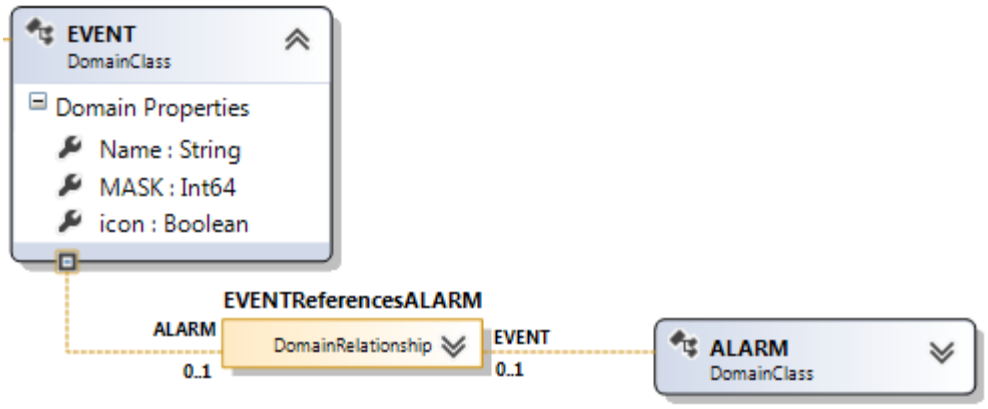


Figura 32 – Modelação do sistema para criação do DSL: EVENT

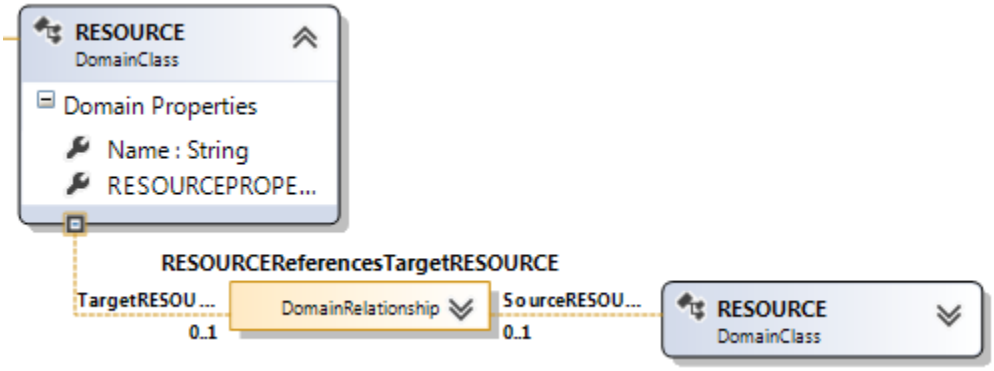


Figura 33 – Modelação do sistema para criação do DSL: RESOURCE

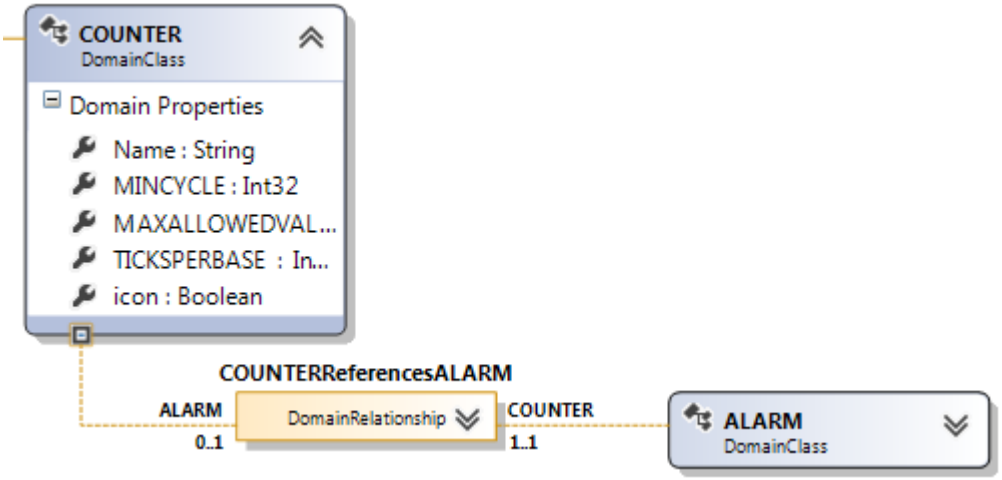


Figura 34 – Modelação do sistema para criação do DSL: COUNTER

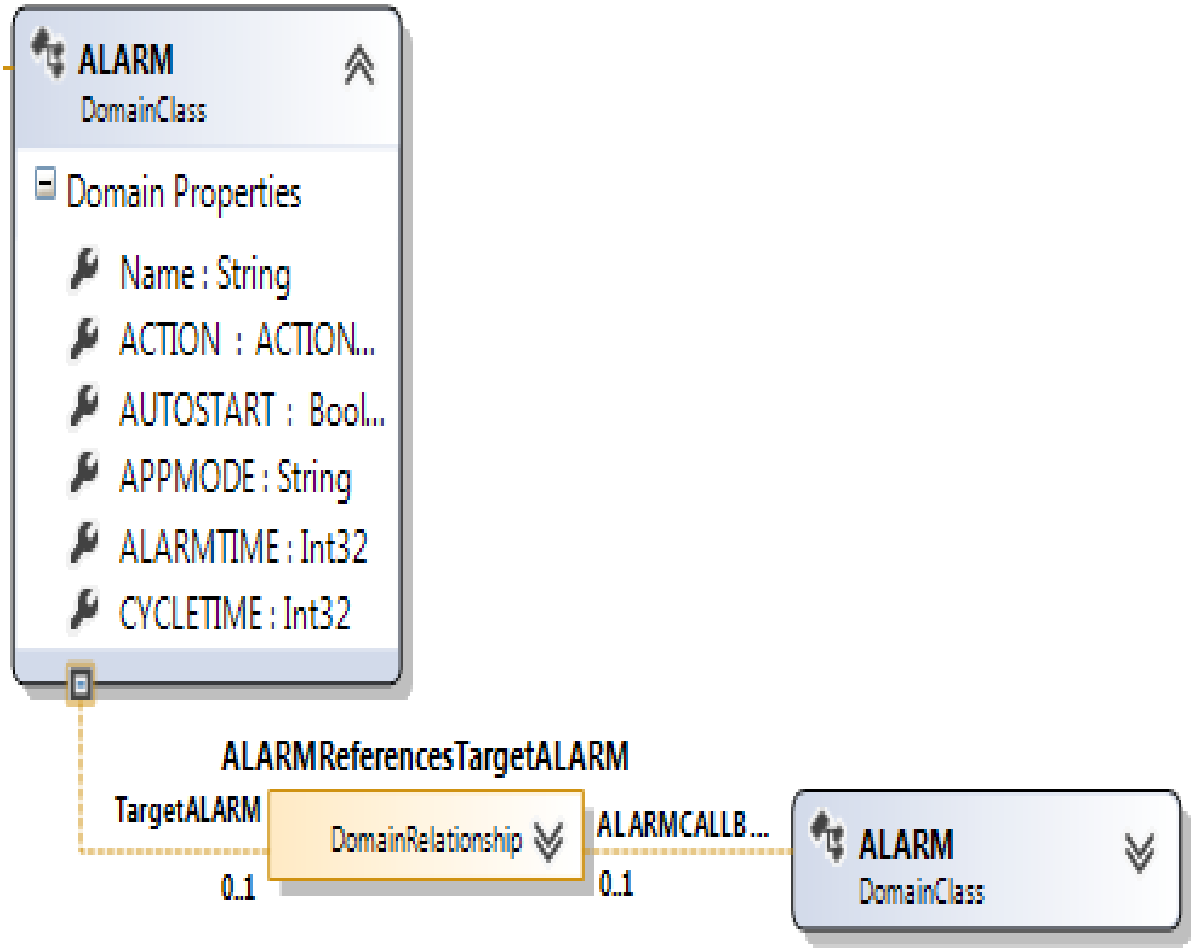


Figura 35 – Modelação do sistema para criação do DSL: ALARM

Graficamente, o CBUILD não apresenta ligação, pois apenas aceita ligações unidirecionais provenientes de outros componentes, como por exemplo, do OSconfig.

Os componentes ISR, EVENT, RESOURCE, COUNTER e ALARM são componentes com ligações diferentes, algumas como no caso do COUNTER-ALARM são de 1..1 e por isso obrigatórias, outras de 0..1 e estão representadas por um campo extra, e também as de 0..* que são feitas através de setas.

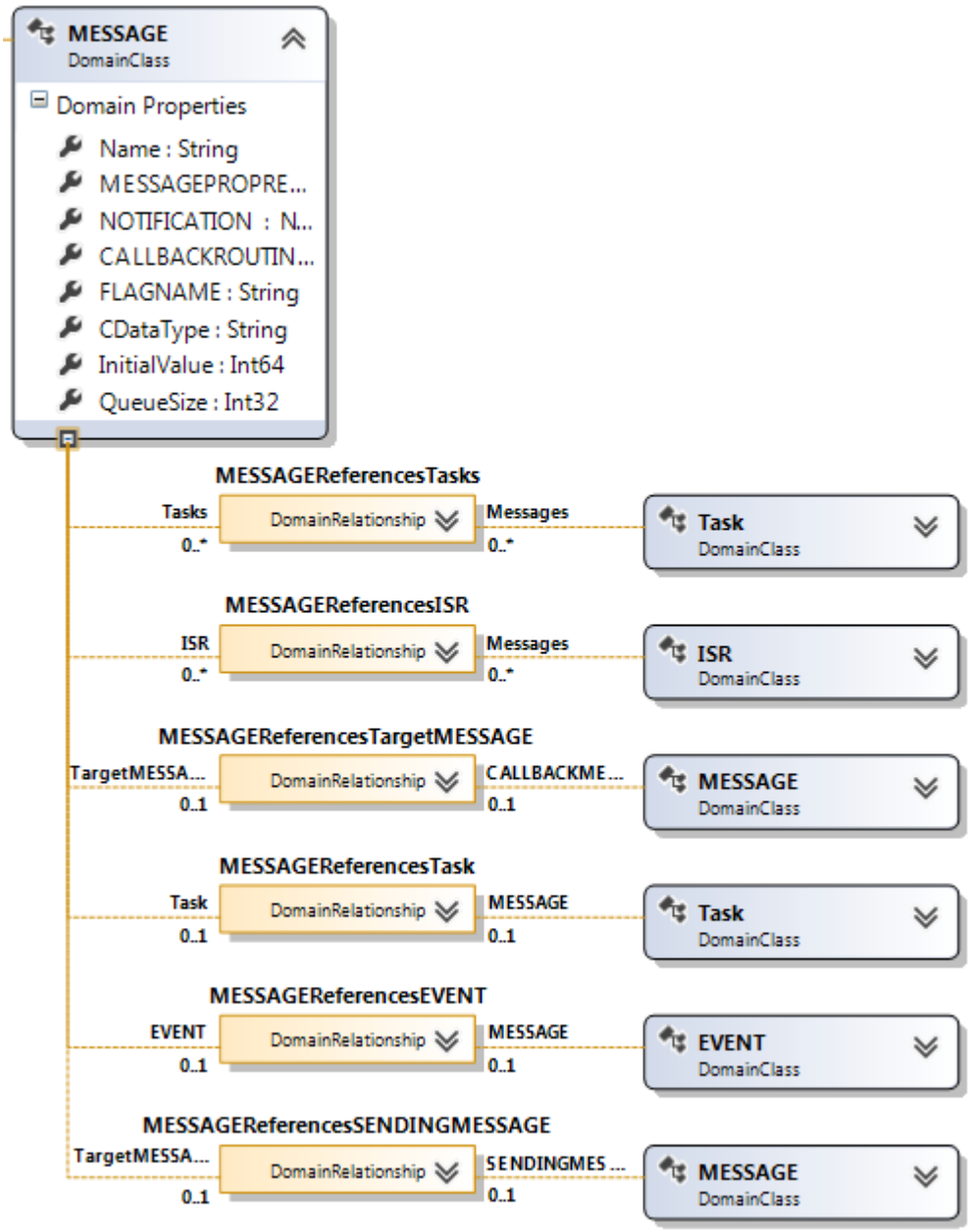


Figura 36 – Modelação do sistema para criação do DSL: MESSAGE

Na implementação de MESSAGE (Figura 36), além dos campos necessários para a sua configuração, temos vários exemplos de ligações, desde as de 0..*, que são representadas por setas, e as de 0..1, em que são usados campos extra para as identificar.

Após a modelação de todos os componentes de acordo com o suporte DSL da *.Net* passa-se à criação das representações dos mesmos que posteriormente serão usados no desenho do sistema.

5.1.2. Criação da representação dos componentes e ligações

Para o desenho do sistema operativo na segunda instância, é necessário que haja um objeto que represente cada um deles. É então preciso criar um objeto para cada componente, um objeto com características únicas para cada componente, que esteja na *toolbox* para se poder aceder no momento do desenho, que tenha uma forma e também algumas características para ajudar na identificação, como nome e símbolo. É também necessário criar as respetivas setas de ligação entre os componentes, como visto anteriormente.

5.1.2.1. Componentes

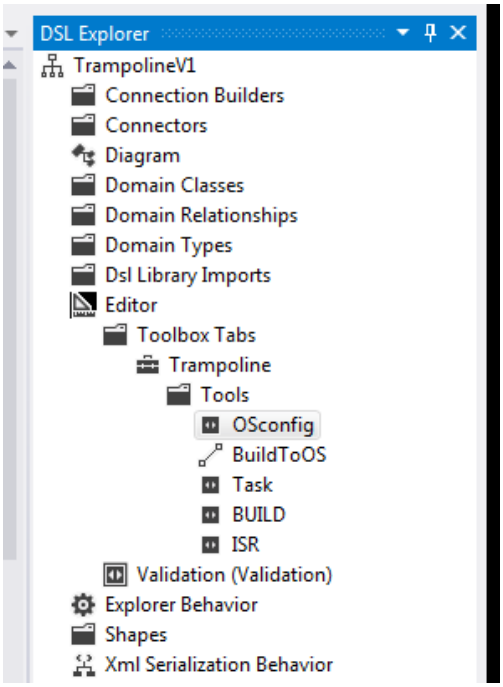


Figura 37 – Componentes: Criação do Componente

Em primeiro lugar é criado na *toolbox* um novo elemento (Figura 37). No momento da criação não está associado a qualquer componente, é apenas um elemento da *toolbox* que está em branco.

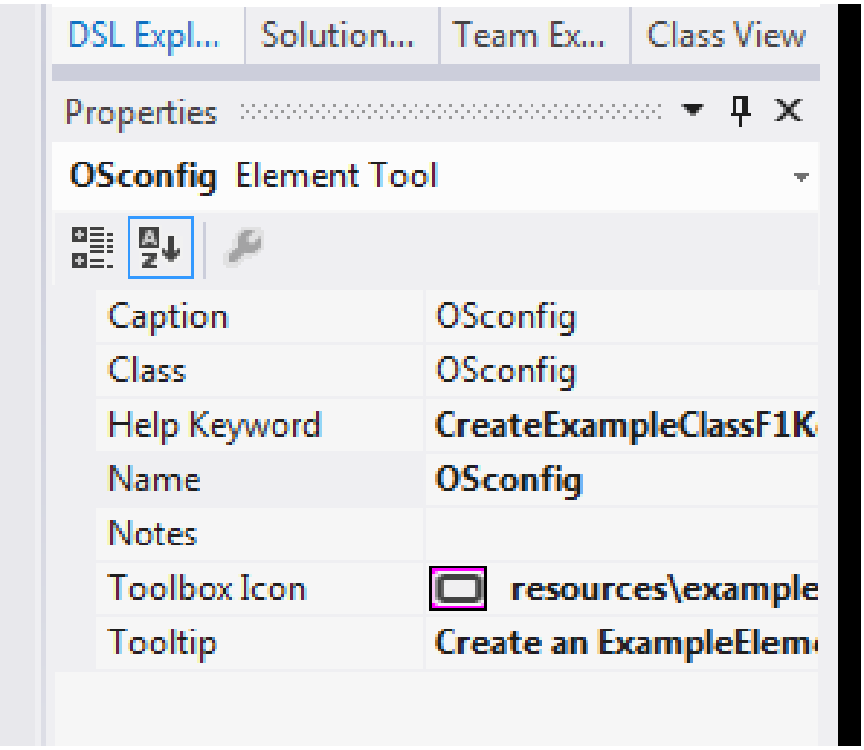


Figura 38 – Componentes: Configuração das Propriedades de componente criado

O novo elemento tem como propriedades os campos visíveis na Figura 38. É então dado ao elemento o mesmo nome do componente que irá representar para uma mais fácil utilização por parte de um utilizador sem formação no IDE agora criado. O elemento é agora associado a uma classe, em outras palavras, associado a um componente específico. É lhe também associada um ícone para o representar na *toolbox*. Os componentes serão todos representados por retângulos.

Depois de criado um elemento na *toolbox* e de o associar a um componente é necessário criar uma representação física do mesmo para usar na área de desenho.

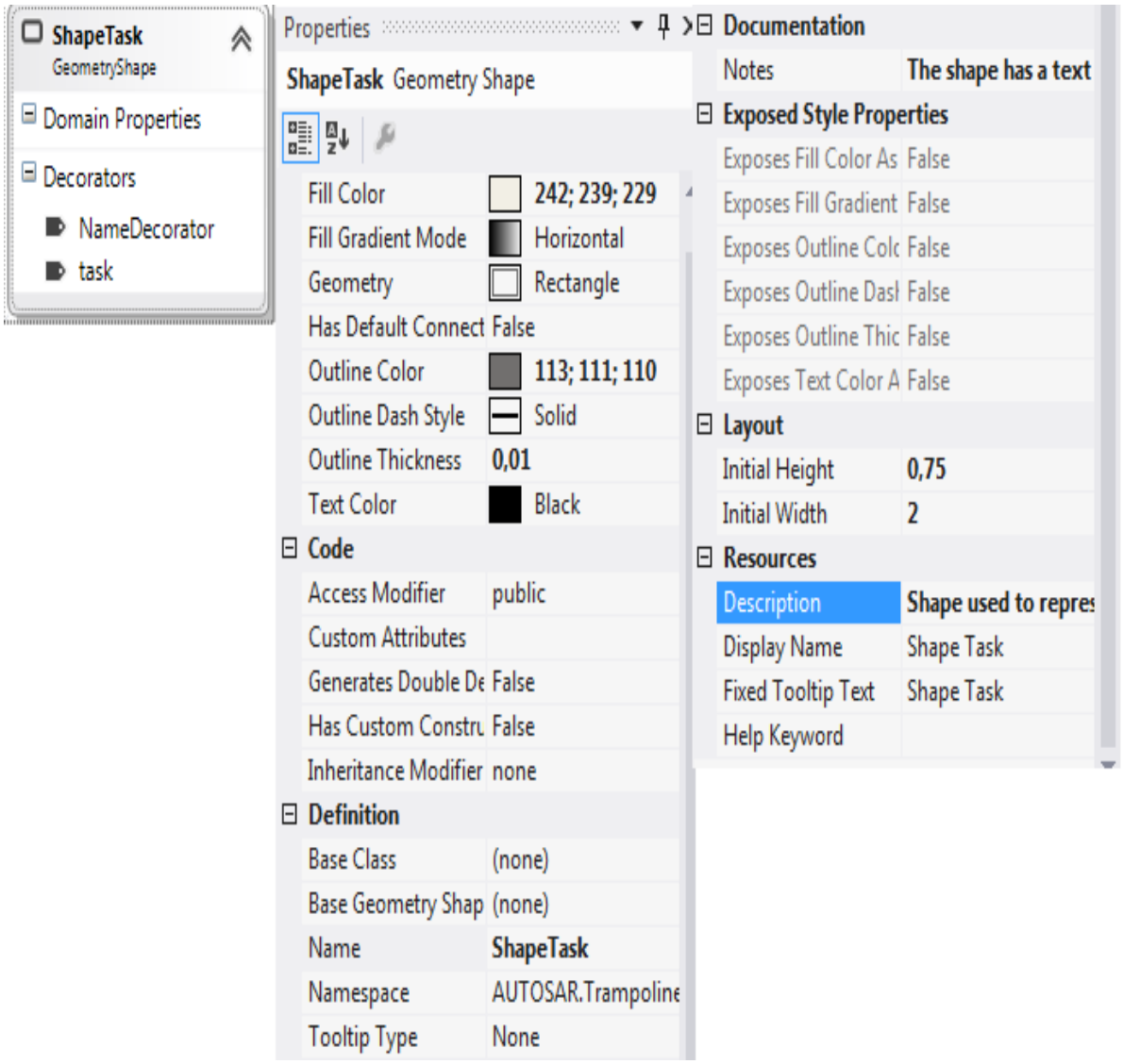


Figura 39 – Componentes: Criação da Forma do Componente

Para a representação do componente na área de desenho é preciso criar uma forma para o mesmo, assim é criado uma *shape* que está associada a um elemento da *toolbox*, e é definida na mesma a forma, cor e tamanho do elemento que será visto na área de desenho (Figura 39).

São também adicionados decoradores ao objeto para uma mais fácil visualização do que o mesmo representa, decoradores estes que permitem que na forma desenhada apareça um nome ou texto a identificar o componente, assim como um decorador com um ícone para identificar o tipo de componente.

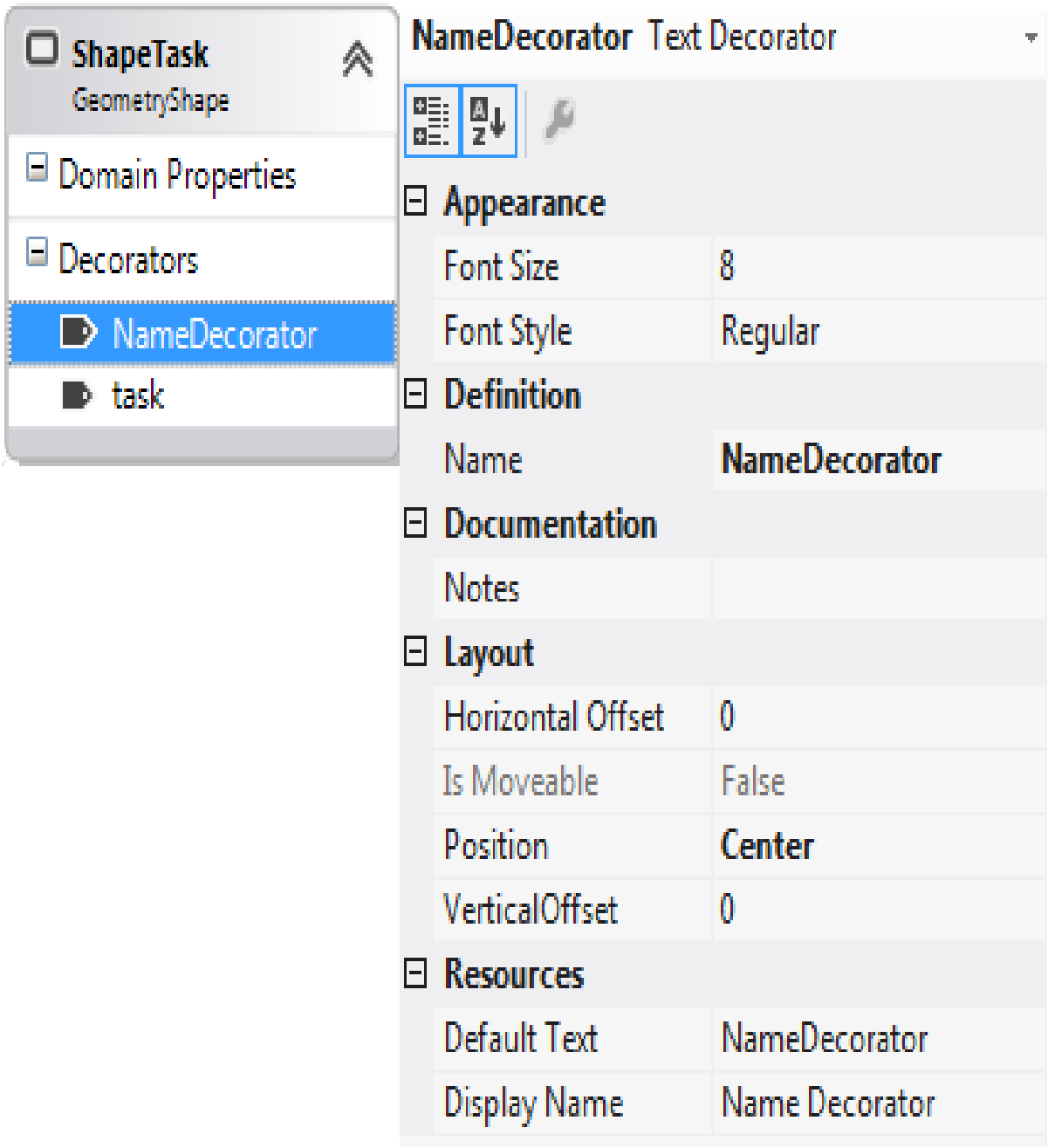


Figura 40- Componentes: Criação da representação do Nome do componente

Na Figura 40 é possível ver o decorador do nome e as suas propriedades, que permite escolher o tipo e tamanho da letra, assim como a sua posição no objeto desenhado. O nome que o objeto apresenta é o mesmo que será dado ao componente no campo NAME do mesmo, que já é definido na modelação como único, e assim não aparecem componentes com nomes iguais, quer no código quer no desenho do sistema.

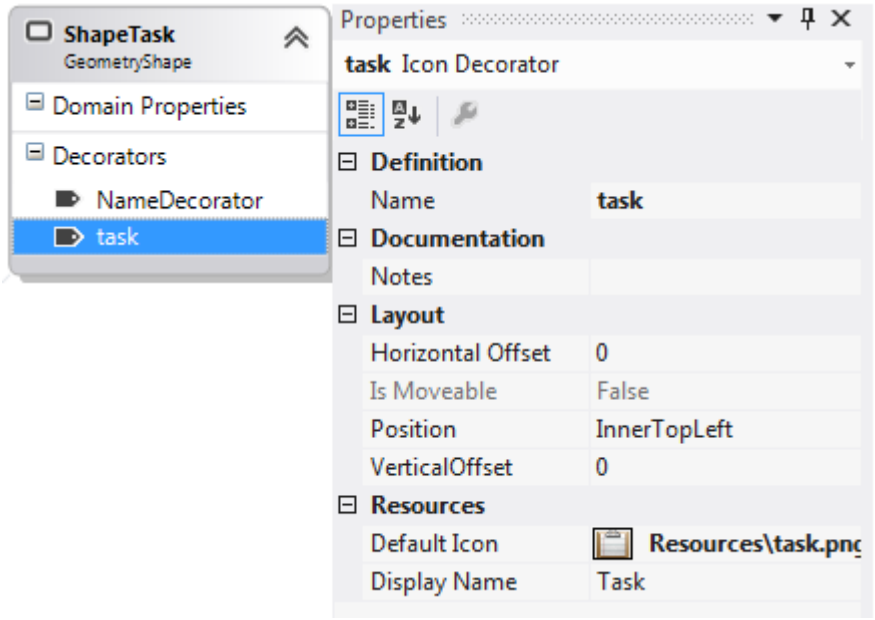


Figura 41 – Componentes: Criação da representação do Ícone do componente

No decorador para o ícone (Figura 41) é que temos uma configuração muito simples. Permite escolher uma imagem para servir de ícone, assim como a sua localização no objeto desenhado.

5.1.2.2. Ligações

Para representar ligações, como já foi explicado anteriormente, diferentes métodos foram utilizados. Para o método que usa um campo extra é apenas necessário acrescentar esse campo na modelação do componente. Para o que é representado por uma seta é necessário criar essas setas e as suas representações.

Para representar as setas é criado na *toolbox* uma nova conexão (Figura 42). Essa nova conexão tem, tal como os elementos, propriedades para atribuir a nova conexão a uma Classe e um ícone com o símbolo que irá representar na *toolbox* para uma fácil identificação e utilização da mesma.

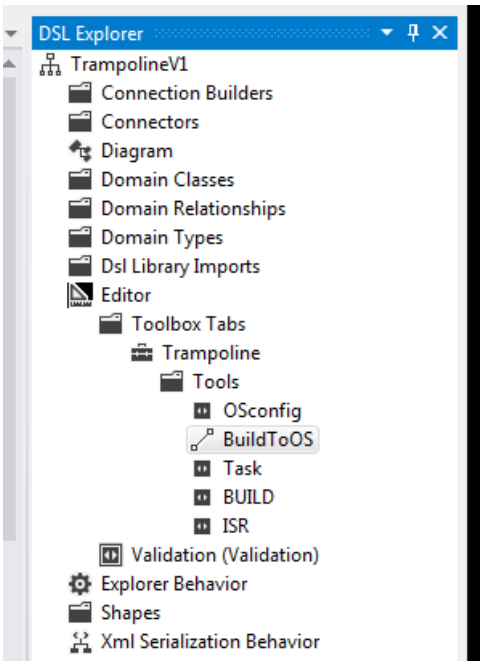


Figura 42 – Ligações: Criação da Conexão

Figuras 42 e 43 apresentam a criação de uma nova conexão na *toolbox* Trampoline, já explicada anteriormente, e tem as seguintes propriedades e configurações:

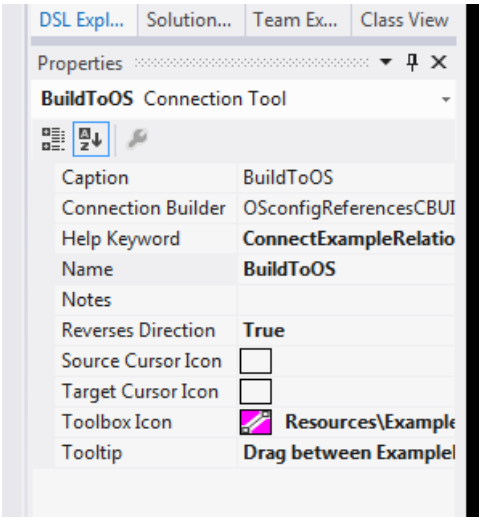


Figura 43 – Ligações: Configuração das Propriedades da Conexão

Na Figura 44 está apresentada a criação da seta como vista pelo utilizador, com todas as suas dimensões e estilos. É assim possível definir toda a estrutura da seta, desde a sua grossura, tipo de traço, tipo de ponta no final, além de como se comporta nos contornos de outros objetos, neste caso usando curvas com ângulos retos.

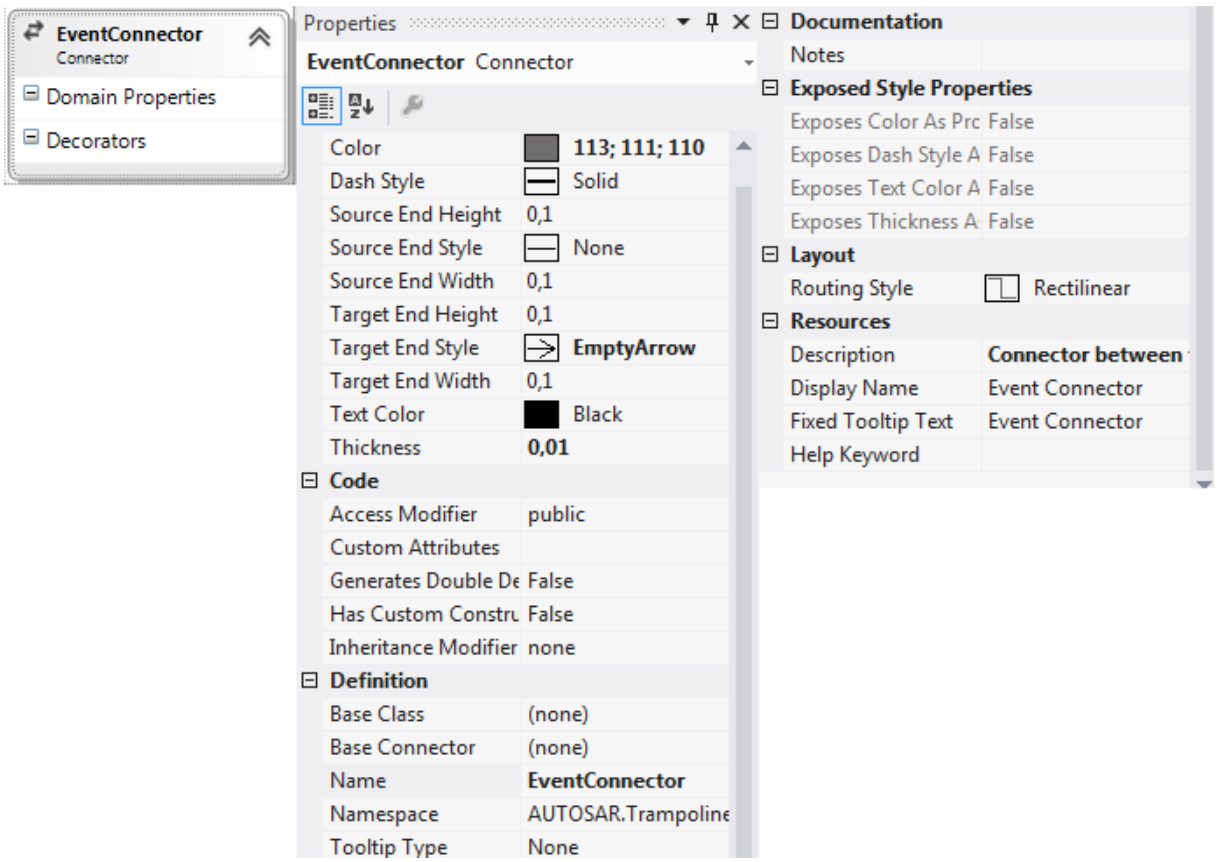


Figura 44 – Ligações: Criação da Forma do conector

5.1.3. Regras de Validação

Como descrito no capítulo 4, são necessárias regras e validações. Para a implementação dessas regras é preciso criar código específico para cada classe para validar a classe e todos os seus campos, sendo também sensíveis a alterações que possam colocar a configuração

incorreta. Para notificar o utilizador é necessário enviar mensagens para o mesmo, mensagens essas que já terão de estar criadas e gravadas no sistema. Por uma questão de fiabilidade e aparência, as mensagens serão enviadas pelo mesmo sistema que o Visual Studio usa na gestão das suas mensagens de erro.

O envio de mensagens de erro através do sistema de gestão de erros do Visual Studio requer a criação de mecanismos designados por recursos que serão descritos nos dois próximos parágrafos.

5.1.3.1. Recursos

Para ser possível apresentar mensagens de erro da forma que foi explicada anteriormente, é necessário criar os mecanismos para isso. Em primeiro lugar é necessário criar um ficheiro do tipo "*resource"* para conter as mensagens de erro necessárias, como se pode ver na Figura 45.

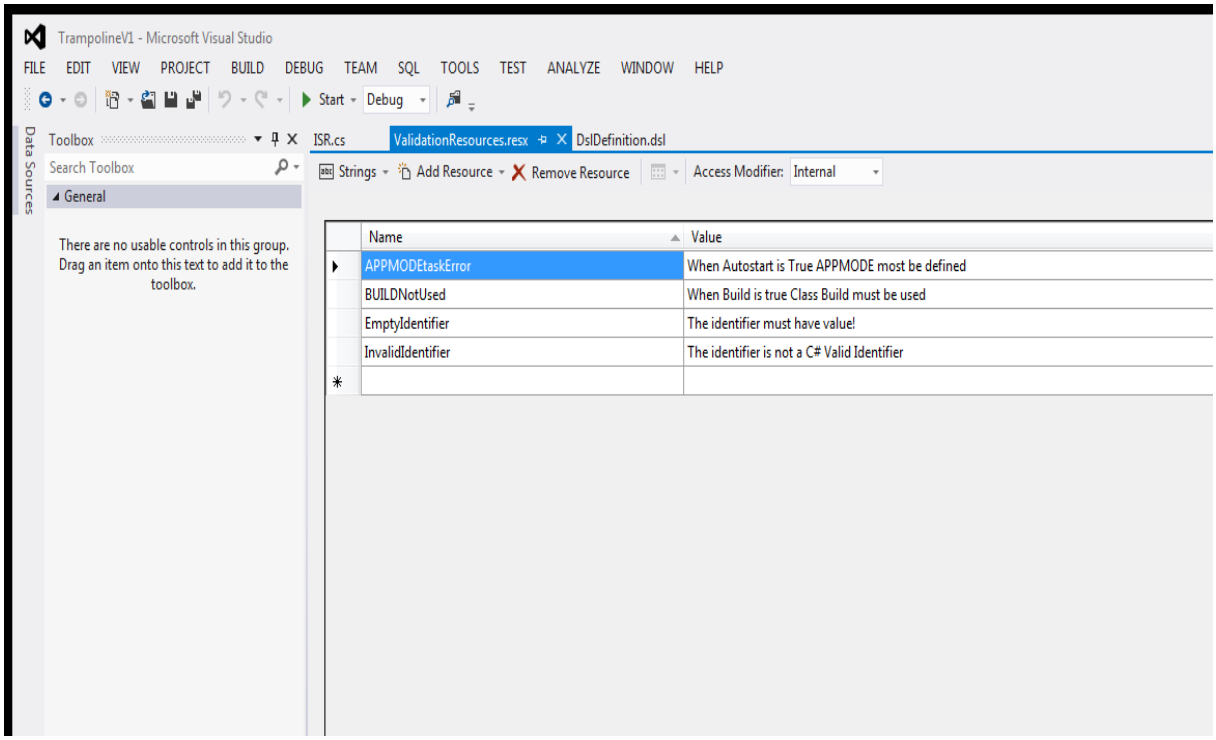


Figura 45 – Validação: Criação dos Mecanismos para apresentação de mensagens de erro

Depois de criadas as mensagens de erro, as mesmas podem ser usadas no código de verificação usando a função de erro fornecida pelo Visual Studio. Com esta função é apenas necessário que, no local onde possam existir erros, se chame a função "*LogError*"(Figura 46) e lhe passe como argumentos a mensagem de erro, o número que o erro irá ter, e o objeto a que o erro é atribuído, como se vê na Figura 46. Tudo isto para que, quando o utilizador experimentar um erro, tenha uma mensagem que o auxilie a perceber o problema, e que se fizer duplo "*click"* na mensagem seja automaticamente encaminhado para o objeto responsável pelo erro, facilitando assim a depuração do mesmo.

```
context.LogError(CustomCode.Validation.ValidationResources.EmptyIdentifier, "Trampoline 005", this);
```

Figura 46 – Validação: Função de mensagens de erro

5.1.3.2. Regras

A verificação de erros é uma parte muito importante deste IDE. Para isso é necessário criar um código específico para cada objeto que irá fazer a verificação de cada objeto e também, quando necessário, de algumas ligações.

Cada objeto tem uma classe que o representa e constrói. Assim, criando ficheiros com "*partial class",* é possível acrescentar código a essas classes. Como o nome indica, uma "*partial class"* permite decompor a implementação de uma classe em vários blocos/ficheiros que serão posteriormente combinados no instante de compilação para formar a classe final. Esta foi a abordagem usada para acrescentar o código de validação a cada objeto, como se pode ver na Figura 47.

```
namespace AUTOSAR.TrampolineV1
{
    [ValidationState(ValidationState.Enabled)]
    public partial class OSconfig
    {
        private static CSharpCodeProvider csharp = new CSharpCodeProvider();

        [ValidationMethod(ValidationCategories.Open | ValidationCategories.Save | ValidationCategories.Menu)]
        private void ValidateAtributeName(ValidationContext context)
        {
            if (string.IsNullOrWhiteSpace(this.Name.Trim()))
                context.LogError(CustomCode.Validation.ValidationResources.EmptyIdentifier, "Trampoline 005", this);
            else if (!csharp.IsValidIdentifier(Name))
            {
                string error = string.Format(System.Globalization.CultureInfo.CurrentUICulture, CustomCode.Validation.ValidationResources.InvalidIdentifier, Name);
                context.LogError(error, "Trampoline 006", this);
            }
        }

        [ValidationMethod(ValidationCategories.Open | ValidationCategories.Save | ValidationCategories.Menu)]
        private void ValidateCBuild(ValidationContext context)
        {
            if (BUILD == true)
            {
                if (CPU.CBUILD == null)
                {
                    string error = string.Format(System.Globalization.CultureInfo.CurrentUICulture, CustomCode.Validation.ValidationResources.BUILDNotUsed, BUILD);
                    context.LogError(error, "Trampoline 007");
                }
            }
        }
    }
}
```

Figura 47 – Validação: Especialização de uma "*Partial Class*"

Na classe parcial é então possível codificar as regras que foram definidas no capítulo 4 para cada um dos objetos. Essas regras são automaticamente verificadas quando o utilizador acede ao menu, ou quando o projeto é:

- Aberto;
- Guardado;
- Compilado.

As regras são implementadas em código C#, usando bibliotecas, métodos e funções fornecidas pelo sistema. Assim é possível fazer as validações, neste caso executando o código de seleção e descriminação do tipo de erro, nos momentos acima referidos, permitindo também enviar as mensagens de erro como descrito no ponto anterior.

5.2. Gerador de Código

O gerador de código é para o utilizador final a parte mais importante, pois é o que lhe dá o que ele precisa/procura, um código pronto a usar. Como explicado no capítulo 4, o gerador de código está dividido em duas partes, a do desenho do sistema e do gerador de código.

O desenho é feito pelo utilizador com a configuração de todos os elementos. Depois na geração de código é esperado que sejam produzidos no final três ficheiros de código. Esse código será gerado usando o desenho do sistema feito pelo utilizador, e usando "*templates*" de codigo, suportados por um DSL específico para o efeito (como explicado no capítulo 3).

5.2.1. Desenho do Sistema

Para o desenho do sistema o utilizador apenas tem de arrastar os componentes e ligações disponíveis na *toolbox* Trampoline (Figura 48)), que foi criada na primeira instância como explicado no ponto anterior.

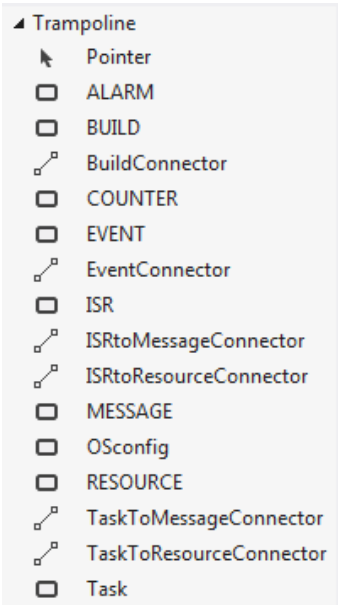


Figura 48 – Geração de Código: *ToolBox* Trampoline

Com os componentes e ligações disponíveis na *toolbox* o utilizador pode criar a sua própria versão do sistema operativo com as funcionalidades que desejar, como o exemplo na Figura 49.

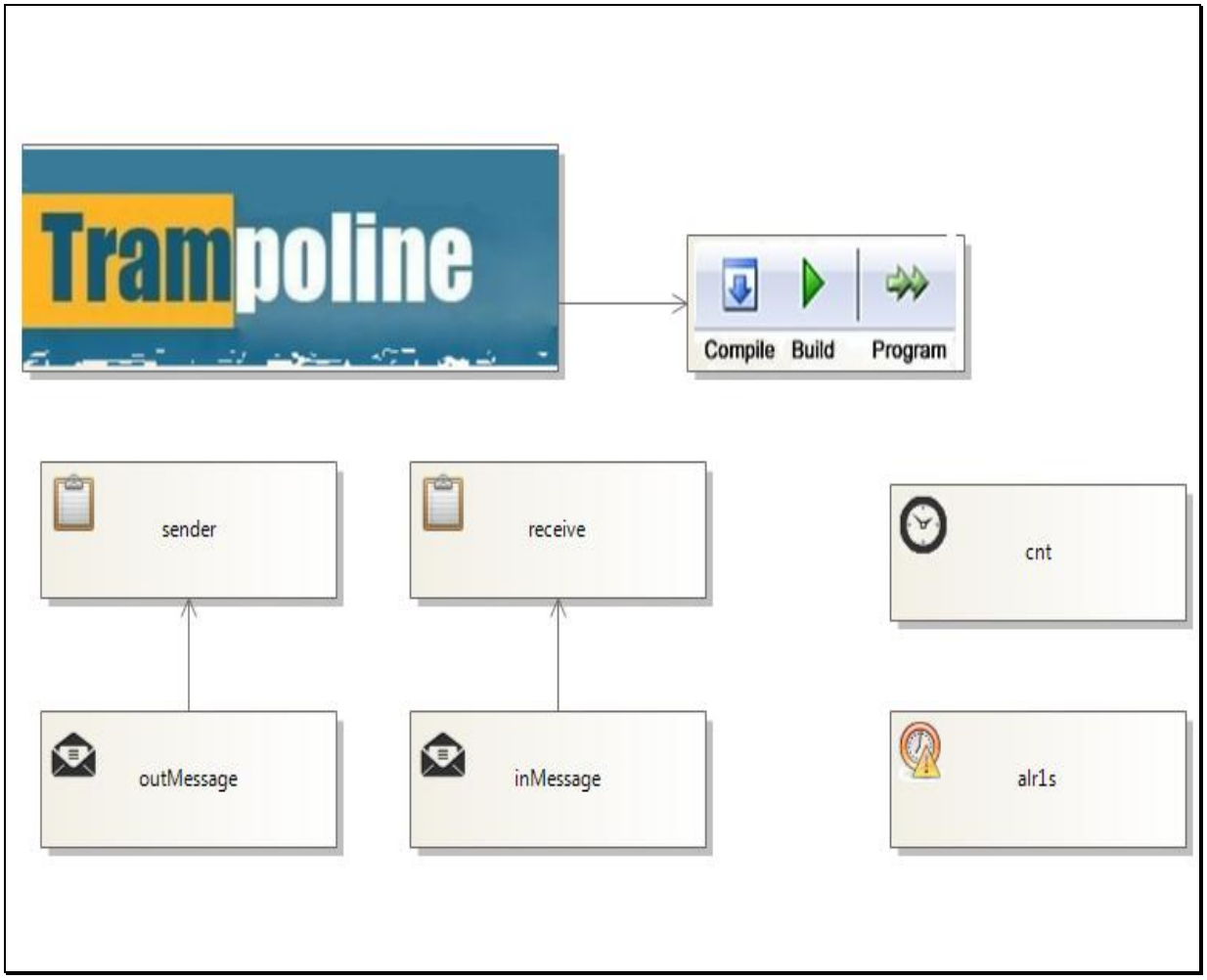


Figura 49 – Geração de Código: Desenho de um sistema

Depois de desenhado cada componente tem a sua própria configuração, como previsto no capítulo 3. A Figura 50 mostra como pode ser configurada um componente TASK. Usando o desenho e as configurações de cada componente é então possível gerar os ficheiros de código.

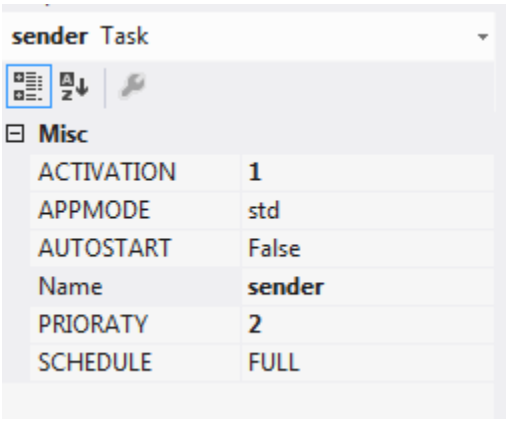


Figura 50 – Geração de Código: Configuração de um componente TASK

5.2.2. Geração de Código

O gerador de código usa um DSL específico do IDE, como já explicado anteriormente, para a criação dos ficheiros de código. Usando as bibliotecas fornecidas pelo Visual Studio e o DSL criado, o processo de criação de ficheiros torna-se mais simples, permitindo a criação de ficheiros com extensões diferentes, como visível nas Figuras 51 e 51.

```
<#@ template inherits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTransformation" #>
<#@ output extension=".c" #>
<#@ TrampolineV1 processor="TrampolineV1DirectiveProcessor" requires="fileName='Sample.tramp'" #>
<#@ assembly name="System.Core.dll" #>
<#@ import namespace="System.Collections.Generic" #>
<#@ import namespace="System.Linq" #>
```

Figura 51 – Geração de Código: Extensão de ficheiro para código C

```
<#@ template inherits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTransformation" #>
<#@ output extension=".oil" #>
<#@ TrampolineV1 processor="TrampolineV1DirectiveProcessor" requires="fileName='Sample.tramp'" #>
<#@ assembly name="System.Core.dll" #>
<#@ import namespace="System.Collections.Generic" #>
<#@ import namespace="System.Linq" #>
```

Figura 52 – Geração de Código: Extensão de ficheiro para código OIL

O *template* de código tem três áreas fundamentais para a criação do código. A primeira área será a de código estático, ou seja, código que será sempre copiado para o ficheiro final sem qualquer alteração, independentemente das configurações usadas, como pode ser visto na Figura 53 marcado a cinzento.

```
#include <stdio.h>
#include "tpl_os.h"
#include "tpl_os_task_kernel.h"

#define _XOPEN_SOURCE 500
#include <unistd.h>

            .
            .
            .

int main(void)
{
    StartOS(OSDEFAULTAPPMODE);
    return 0;
}
```

Figura 53 – Geração de Código: Código estático simples

Mas nem todo o código a cinzento está sempre presente, pois algum encontra-se entre código de seleção, como é o caso da função "*StartupHook"* (Figura 54), que só é usada quando é configurada como "*True"*, mas quando está configurada para ser usada o seu código é sempre igual.

```
#include <stdio.h>
#include "tpl_os.h"
#include "tpl_os_task_kernel.h"

#define _XOPEN_SOURCE 500
#include <unistd.h>

<#
    foreach (Task task in this.CPU.Tasks)
    {
#>
TASK(<#=            task.Name #>);
<#    } #>

<#
    foreach (MESSAGE task in this.CPU.Messages)
    {
#>
DeclareMessage(<#=            task.Name #>);
<#    } #>

int main(void)
{
    StartOS(OSDEFAULTAPPMODE);
    return 0;
}
<#    if(this.CPU.OSconfigs.STARTUPHOOK == true){ #>
void StartupHook(void)
{
}
<#    }#>
```

Figura 54 – Geração de Código: Exemplo de código estático complexo

Como segunda área, existe a de código dinâmico, em que o código que irá aparecer no ficheiro final será o mesmo que o utilizador escrever nas configurações. Como se pode ver na Figura 55, existe a "*Flag" "APP_SRC"* que é estática, o valor que recebe é dinâmico, depende do que o utilizador escrever na configuração do sistema, e assim o código que será escrito no ficheiro de código final irá depender da configuração selecionada no momento da sua geração.

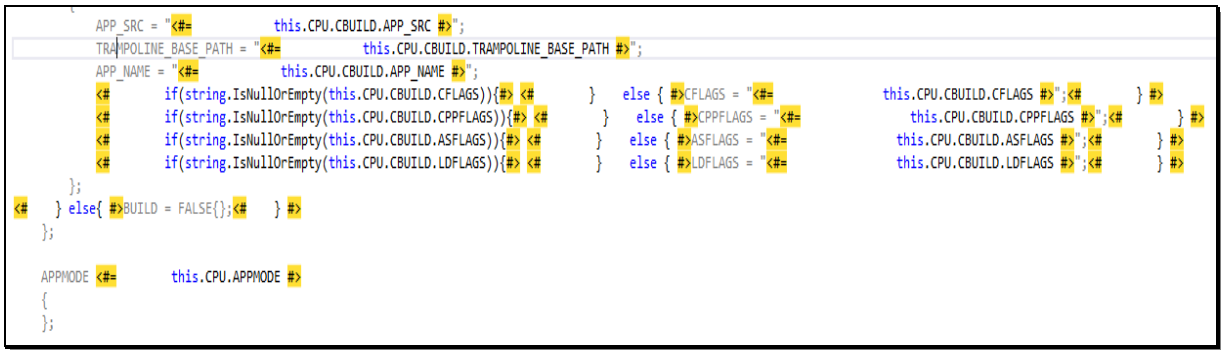


Figura 55 – Geração de Código: Código dinâmico

A terceira e última área é a área das funções internas (Figura 57), funções que serão usadas para a criação de código para o ficheiro final. Como visto no capítulo 4, uma tarefa pode estar associada a vários eventos, e por isso, como se pode ver nos exemplos seguintes, é preciso listar todos os eventos associados a uma tarefa. Para tal serão usadas duas funções internas: a primeira "*enumeventos*", que cria uma lista com todos os eventos ligados a uma determinada tarefa, e a segunda função "*ListarEventos*" para escrever no ficheiro o código necessário (Figura 56).

Figura 56 – Geração de Código: Exemplo da Chamada das funções internas

```
<#+
    IEnumerable<EVENT> enumeventos(Task task)
    {
        foreach(EVENT ev in TaskReferencesEvents.GetEvents(task))
        {
            yield return ev;
        }
    }

    void ListarEventos(IEnumerable<EVENT> ev)
    {
        foreach(EVENT listar in ev)
        {
            string print ="         "+"EVENT = "+ listar.Name + ";\n";
            Write(print);
        }
    }
```

Figura 57 – Geração de Código: Exemplo da implementação de funções internas

Com tudo o que foi explicado até este ponto serão criados os três "*templates*", um para cada ficheiro de código esperado, e assim gerado o código final esperado.

Capítulo 6

TESTES E RESULTADOS

Neste capítulo serão apresentados os resultados produzidos nesta dissertação. Os resultados serão apresentados sobre a forma de três testes. Os testes servirão para provar que o IDE é capaz de produzir código executável e o IP do sistema, cumprindo assim o proposto.

O autor optou por apenas apresentar estes três testes, sendo que será primeiro apresentado um teste simple, depois um teste de nível intermédio, e por último um teste mais complexo.

No primeiro teste serão demonstrados todos os passos para a criação da aplicação, nos outros dois apenas os passos mais importantes e frequentemente utilizados serão mostrados, sendo assim ocultados os passos que são sempre repetidos.

6.1. Teste Simples

O primeiro teste trata-se de um teste muito simples, é apenas esperado que o IDE seja capaz de gerar uma aplicação composta por duas tarefas, com prioridades diferentes. É esperado que a tarefa de mais baixa prioridade inicie automaticamente quando o sistema arrancar, mostre uma mensagem para demonstrar que iniciou, e depois que ative a tarefa de prioridade superior. Essa tarefa enviará também uma mensagem, e depois termina, voltando o sistema para a tarefa de prioridade inferior, implementando um ciclo para criar algum tempo de atraso antes de repetir de novo todo o processo.

As figuras seguintes servirão para ilustrar todo o processo de teste e para provar a veracidade do mesmo.

A Figura 58 ilustra o desenho feito no IDE para criar a aplicação proposta.

Figura 58 – Teste Simples: Desenho da Aplicação

Cada componente tem de ser configurado de acordo com as especificações. A Figura 59 ilustra esse processo assim como mostra os valores usados.

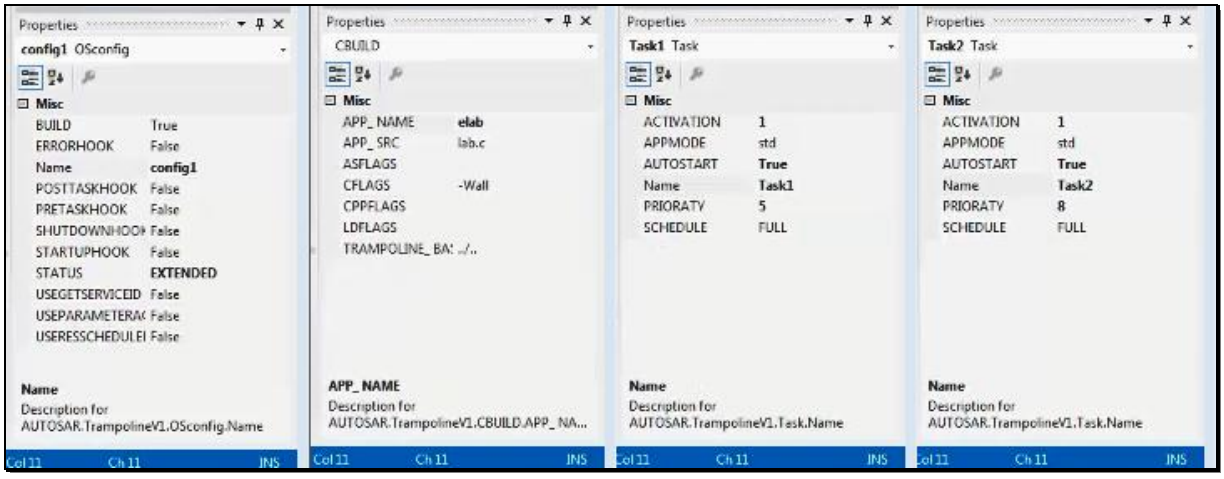


Figura 59 – Teste Simples: Configuração das Propriedades

Após o desenho e configuração do sistema é feita a compilação, ou como o VMSDK chama, transformação das *templates*, para criar os ficheiros, como mostra a Figura 60.

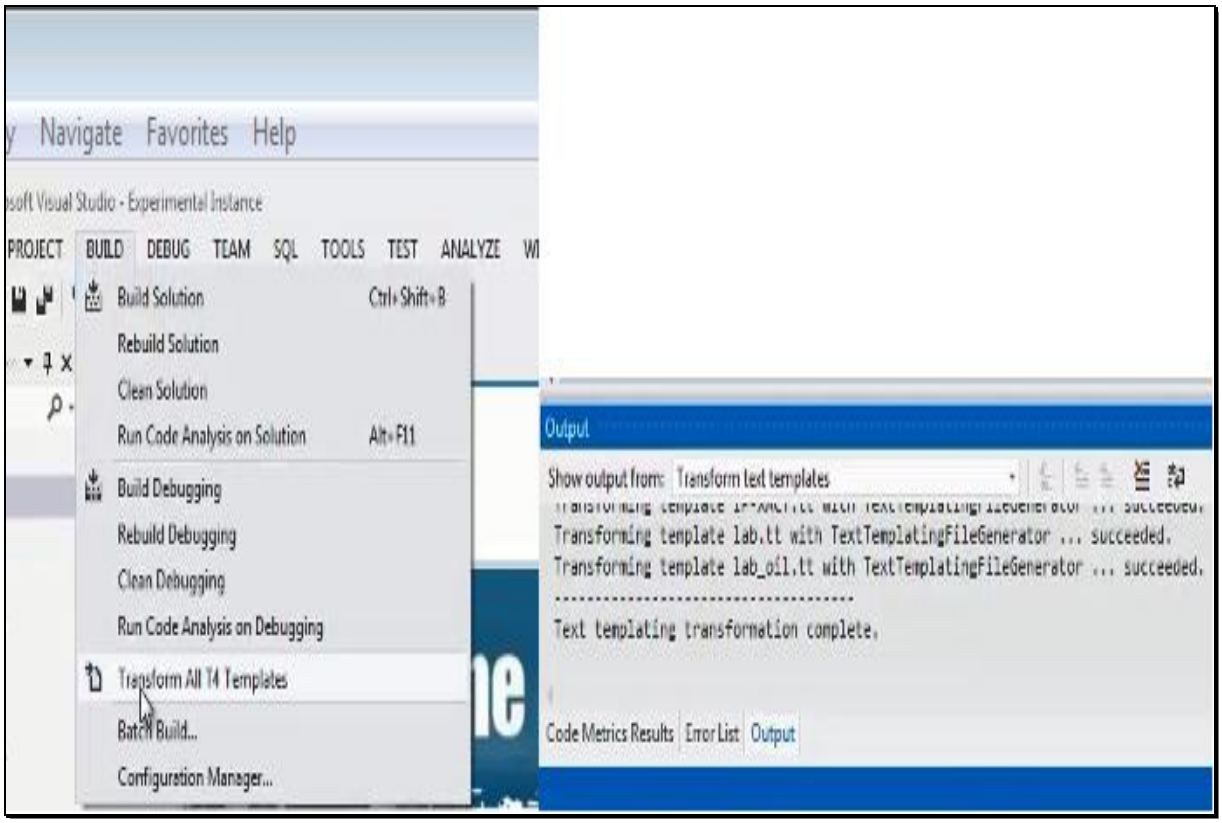


Figura 60 – Teste Simples: Transformação das *templates*

Acabando a transformação dos "*templates"*, os ficheiros de código já estão disponíveis ao utilizador com o código que o mesmo precisa, o Código C e OIL, sendo apenas preciso agora acrescentar o código de cada tarefa. Além desses códigos também o ficheiro XML com o IP do sistema foi gerado (Figura 61). As imagens seguintes vão demonstrar o código criado e o código que o utilizador teve de escrever manualmente.

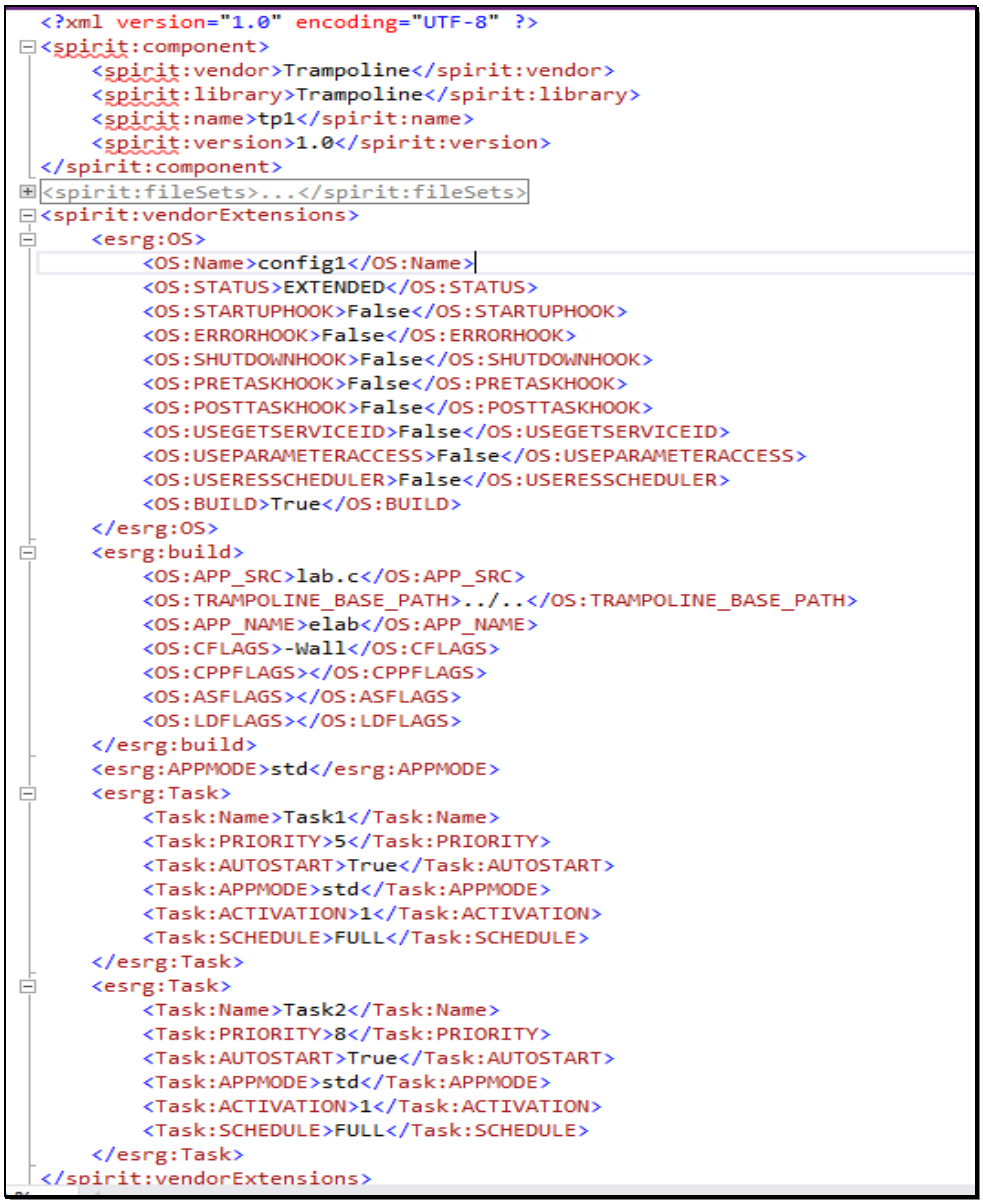


Figura 61 – Teste Simples: Excerto do XML com o IP do sistema

O segundo ficheiro contém o código OIL gerado, o mesmo já se encontra pronto a compilar (Figura 62).

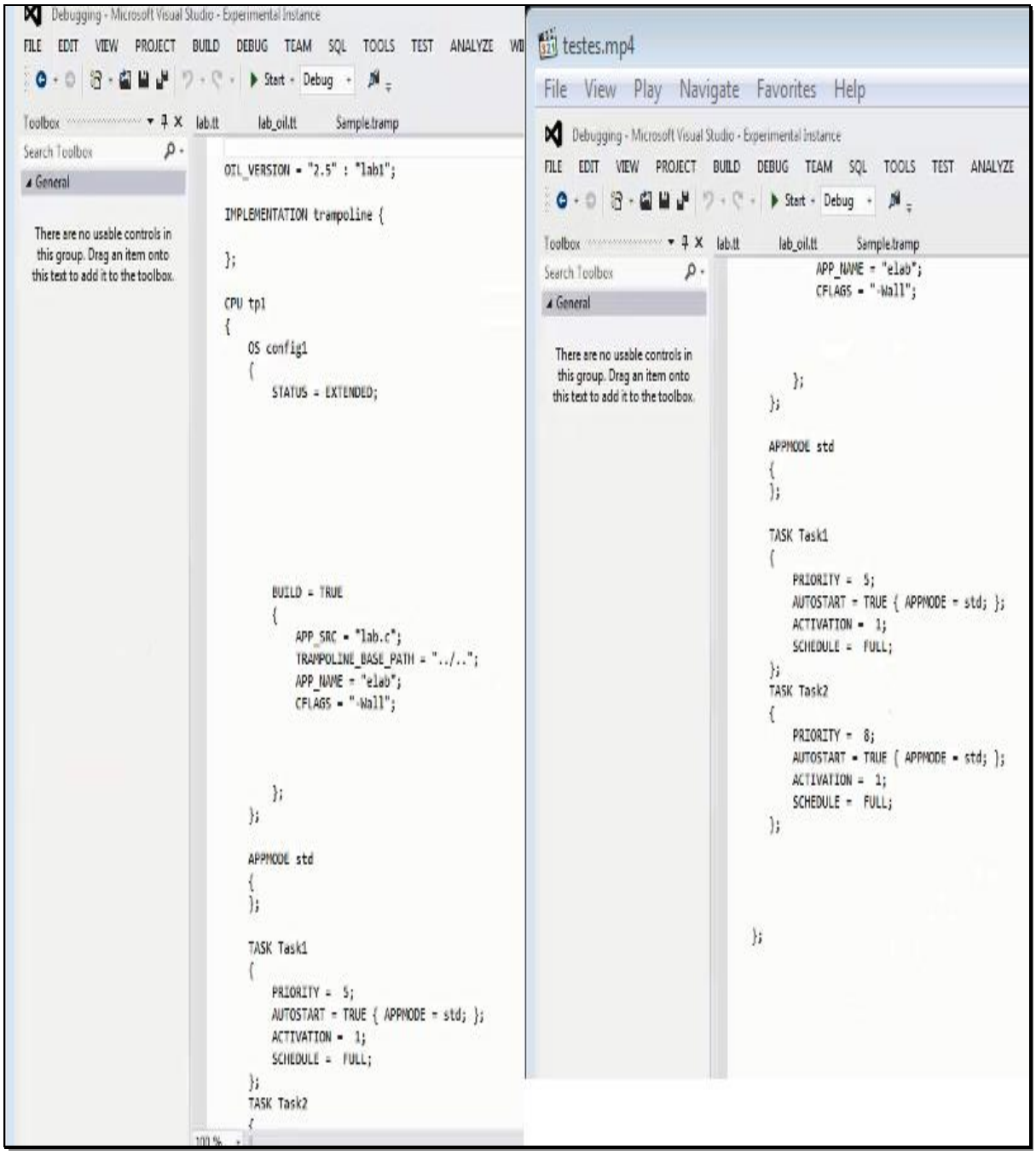
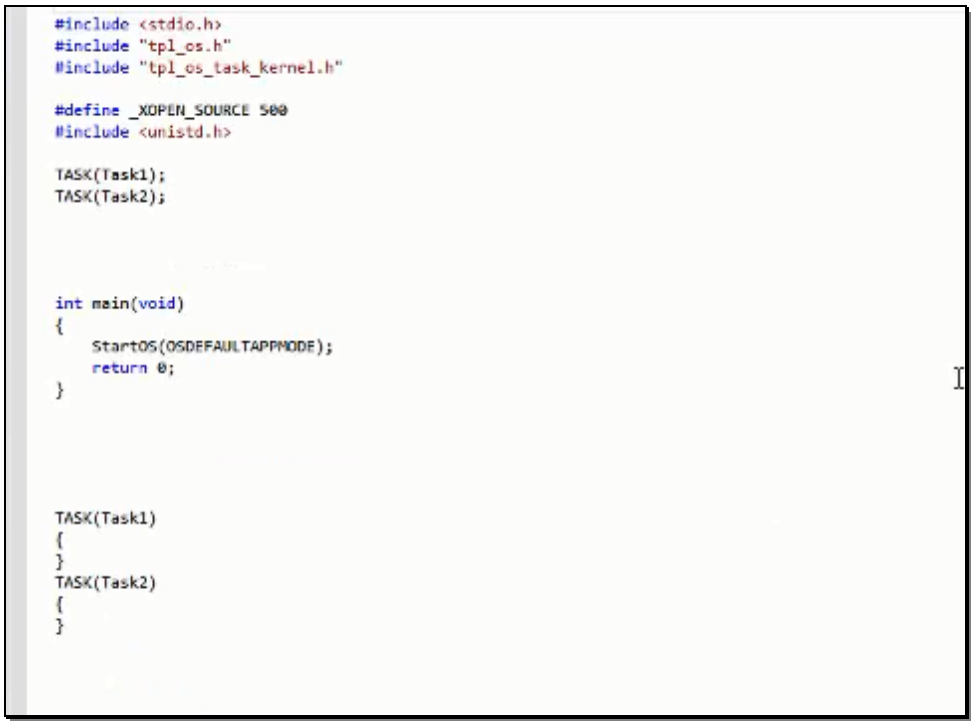


Figura 62 – Teste Simples: Código OIL

Por fim o terceiro ficheiro com o esqueleto do código C. A Figura 63 ilustra o esqueleto do código, enquanto que a Figura 64 ilustra o código de um possível comportamento pretendido para o sistema.



```c
#include <stdio.h>
#include "tpl_os.h"
#include "tpl_os_task_kernel.h"

#define _XOPEN_SOURCE 500
#include <unistd.h>

TASK(Task1);
TASK(Task2);



int main(void)
{
    StartOS(OSDEFAULTAPPMODE);
    return 0;
}




TASK(Task1)
{
}
TASK(Task2)
{
}
```

Figura 63 – Teste Simples: Esqueleto do Código Gerado

```c
23 TASK(Task1)
24 {
25     unsigned int i;
26     i = 0;
27     while(1==1)
28     {
29     i++;
30     printf("Task 1, chama 2.\n");
31
32     ActivateTask(Task2);
33     for(i=0;i<10E7; i++);
34     }
35 }
36 TASK(Task2)
37 {
38     printf("Sou a mais prioritaria, task 2\n");
39     TerminateTask();
40 }
41
```

Figura 64 – Teste Simples: Possível comportamento do Sistema

Depois de todo o código estar pronto só é preciso compilar e executar. A compilação será em duas partes, a primeira é a compilação do ficheiro OIL invocando o compilador GOIL e só

depois é invocado o compilador C. A figura 65 ilustra a compilação do GOIL, que só devolve mensagens se houver erros, senão apenas compila e termina.

Figura 65 – Teste Simples: Compilação dos ficheiros OIL usando GOIL

Com a compilação do OIL concluída, um *MakeFile* é criado e passado ao comando "make" para gerar o ficheiro executável, como mostram Figuras 66 e Figura 67.

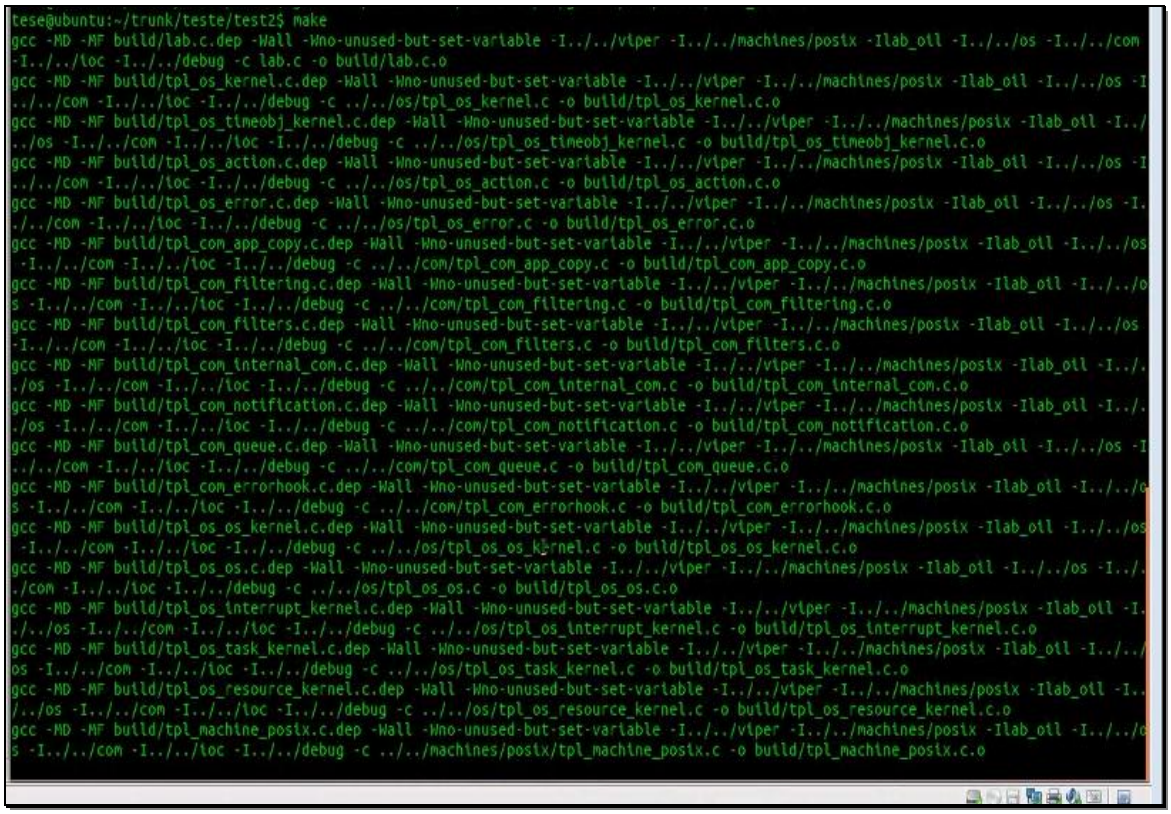


Figura 66 – Teste Simples: Compilação C Parte 1

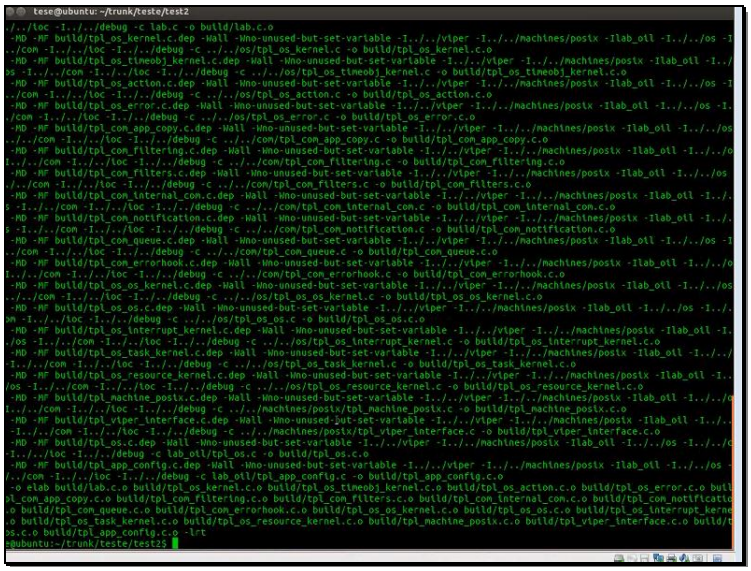


Figura 67 – Teste Simples: Compilação C Parte 2

Por fim é possível executar a aplicação. A Figura 68 exibe a execução da aplicação proposta para este teste.

Figura 68 – Teste Simples: Execução da Aplicação

6.2. Teste Intermédio

O segundo teste já é um pouco mais complexo, continua a ter só duas tarefas, mas desta vez quando o sistema inicia o que é inicializado automaticamente é um contador que ativa uma tarefa em intervalos periódicos. A tarefa ativada envia uma mensagem para outra tarefa, que é

ativada quando recebe a mesma, imprime-a no ecrã e termina. Assim é possível ilustrar o sistema de troca de informações entre tarefas usando mensagens.

Então será feito primeiro o desenho do sistema e a configuração dos componentes, como demonstram a Figuras 69 e a Figura 70.

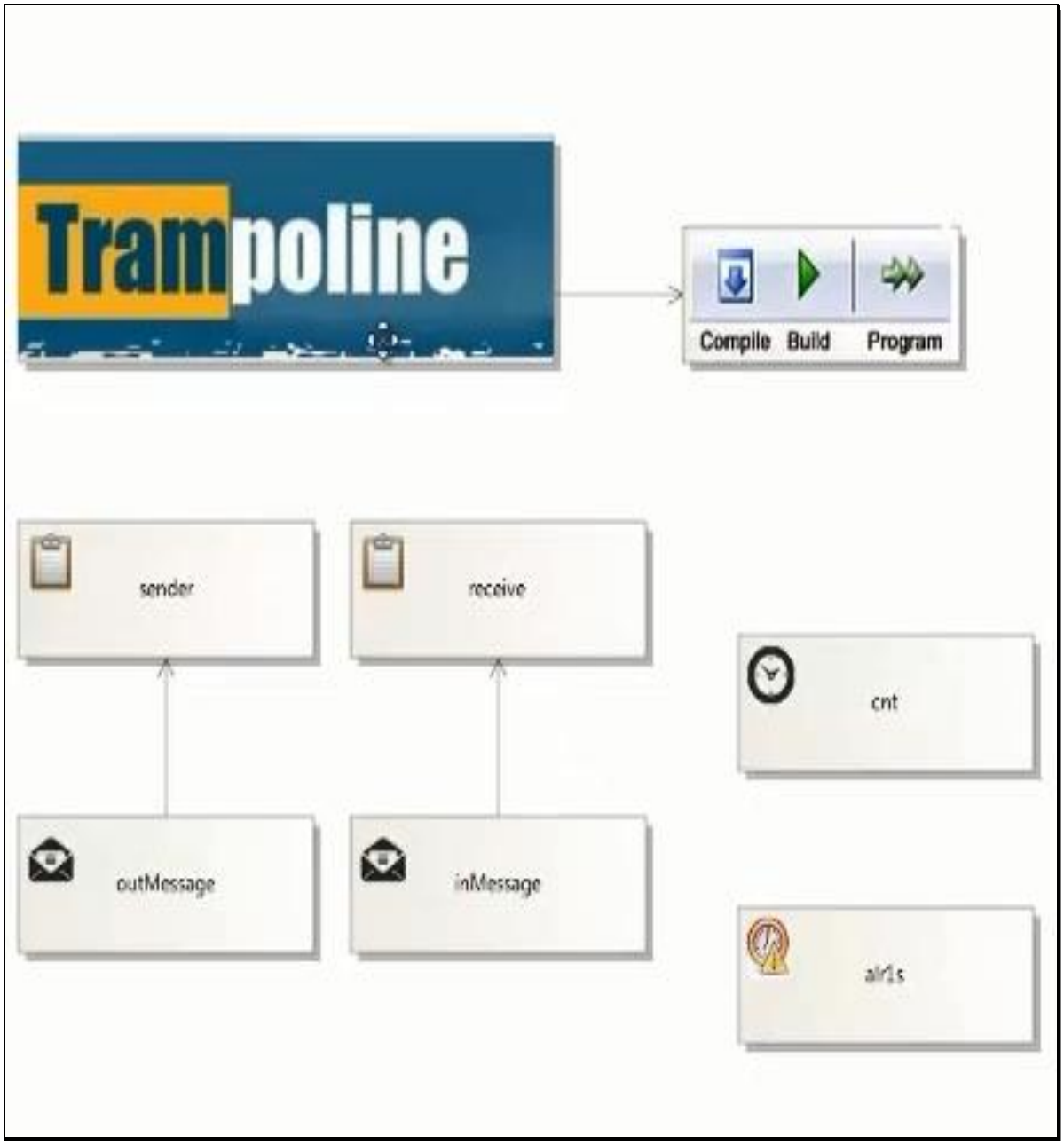


Figura 69 – Teste Intermédio: Desenho do Sistema

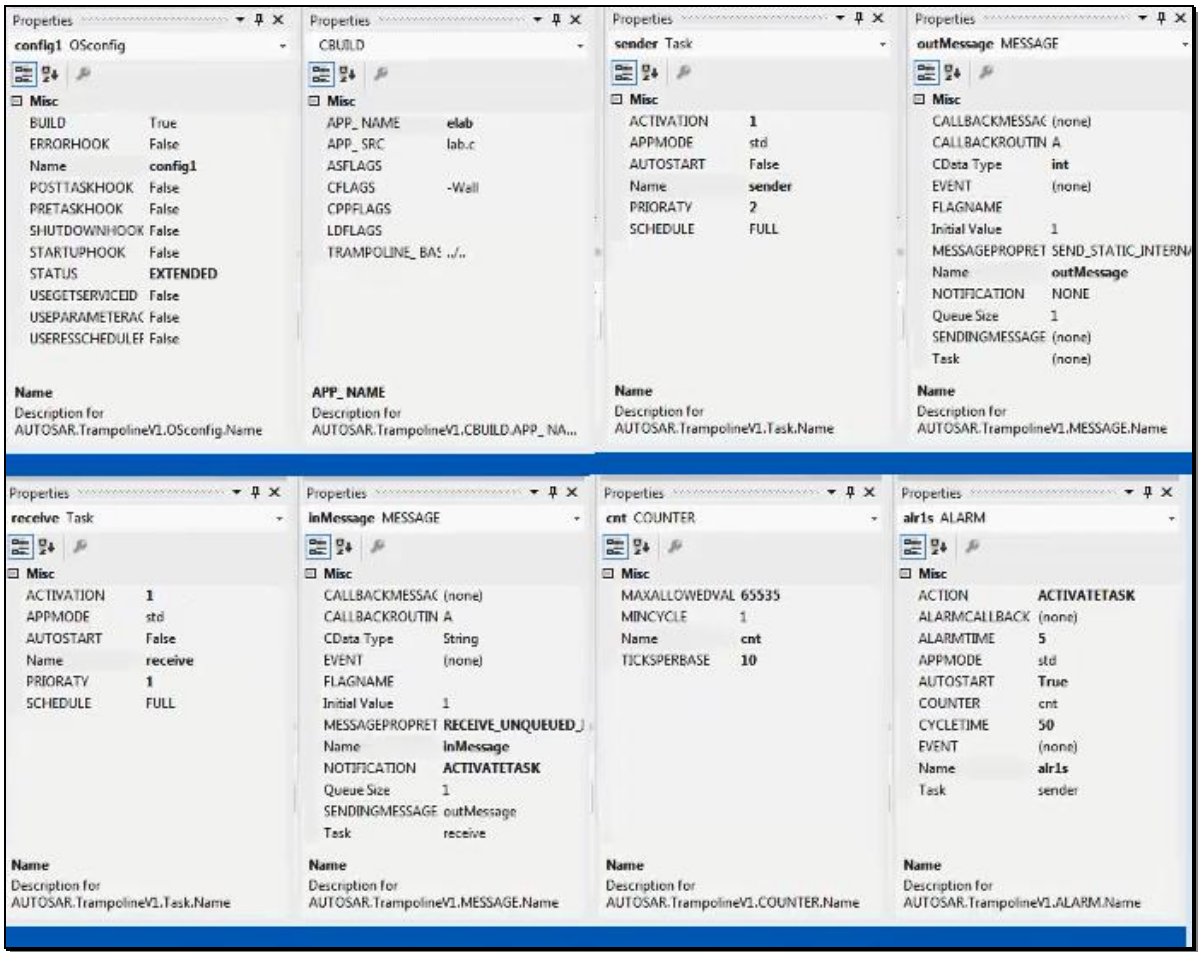


Figura 70 – Teste Intermédio: Configuração das Propriedades

Depois de configurado e desenho do sistema, segue-se a geração dos ficheiros. Como é um processo igual ao do ponto anterior não são exibidas imagens do processo. Depois de gerados, o utilizador tem o código à sua disposição e só precisa de completar o código C das tarefa.

A Figura 71 mostra o código XML do IP desta nova aplicação criada. O código IP é parecido com o do teste anterior pois ambas as aplicações usam o mesmo nome e número de versão, mas a descrição dos componentes que compõem esta aplicação é única e diferente de qualquer outra.

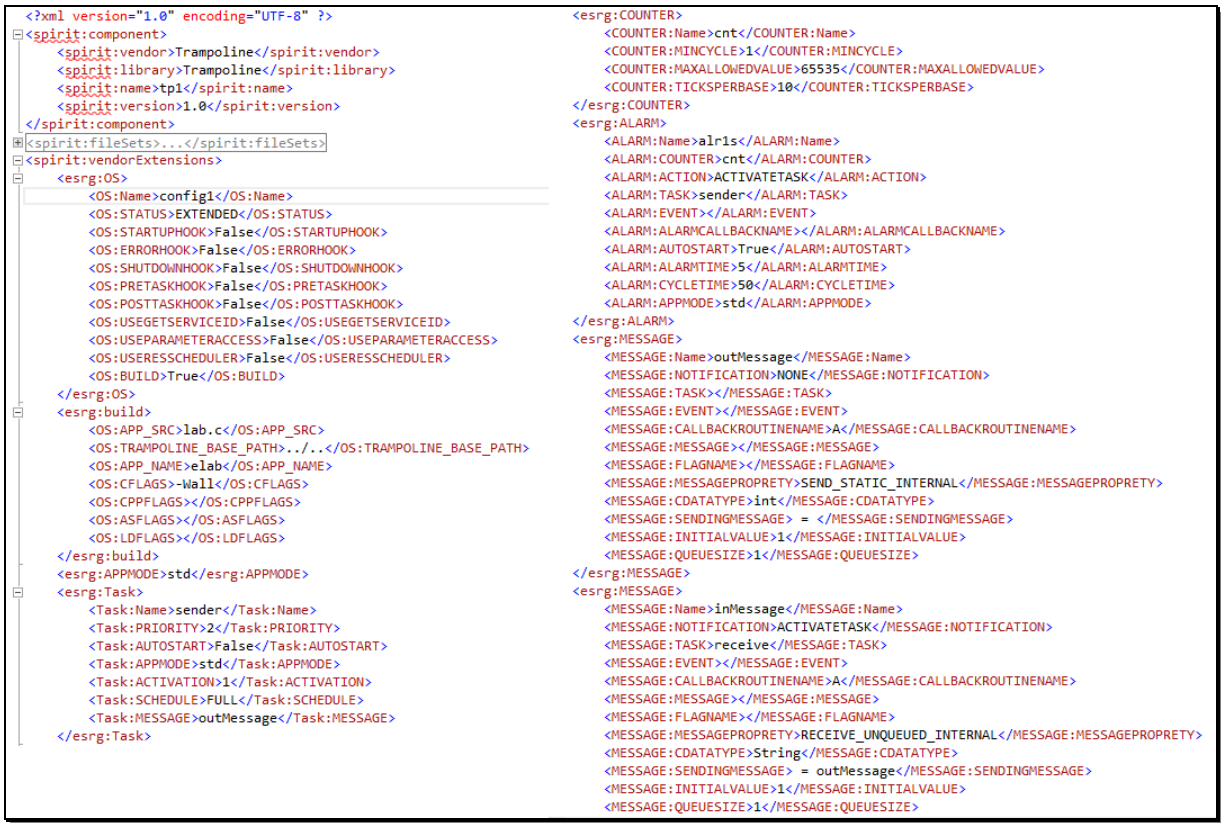


Figura 71 – Teste Intermédio: Excerto do XML com o IP do Sistema

A Figura 72 mostra o código OIL gerado para este teste que está pronto a ser compilado pelo GOIL.

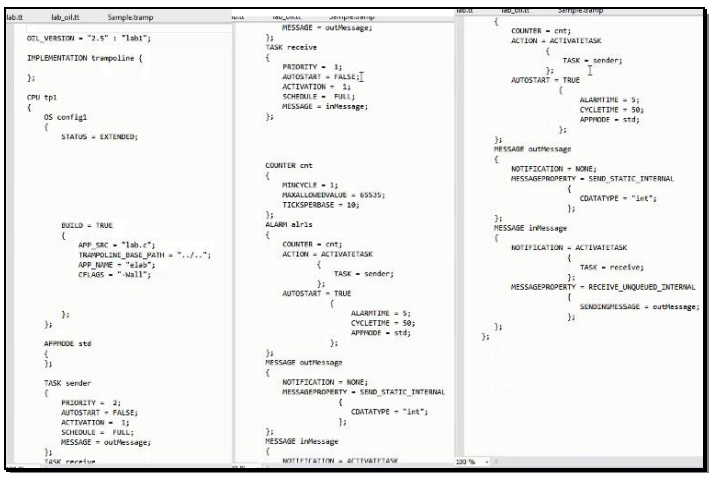


Figura 72 – Teste Intermédio: Código OIL

Ao código C gerado só é preciso que o utilizador acrescente o código comportamental especializado para cada tarefa. A Figura 73 mostra o código gerado e o que foi acrescentado, para incrementar uma variável numérica e enviá-la para a outra tarefa, demonstrando assim que o valor vai incrementando entre envios.

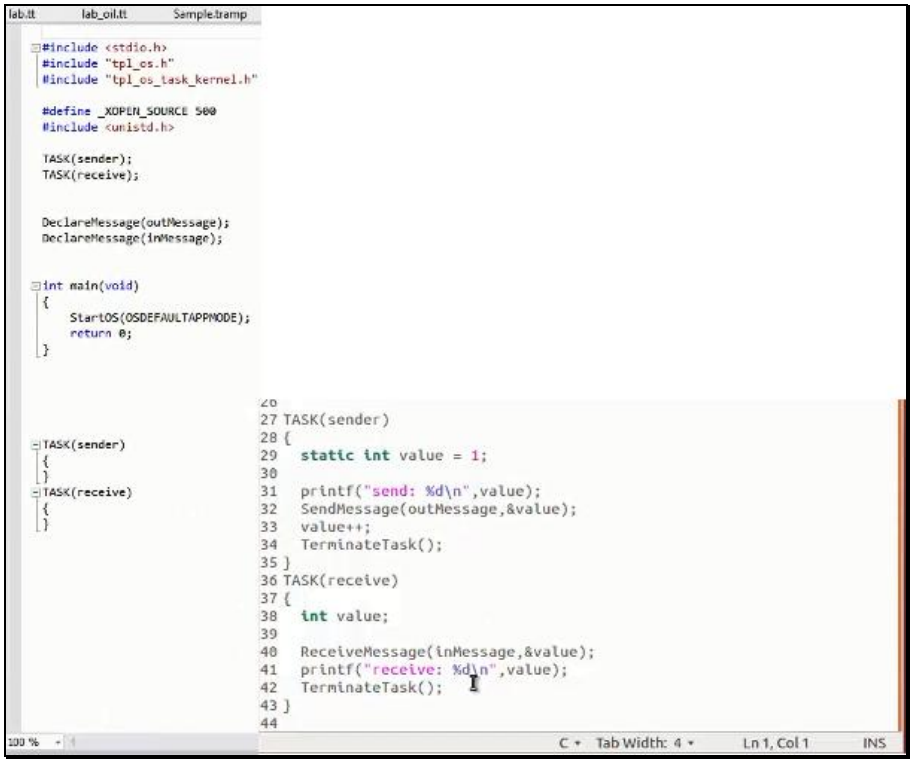


Figura 73 – Teste Intermédio: (Esquerda) Esqueleto do Código Gerado (Direita) Possível Comportamento do Sistema

Depois de compilar os ficheiros OIL e C, a aplicação está pronta a executar. O segundo teste é comprovado pela Figura 74, que mostra a execução da aplicação.

Figura 74 – Teste Intermédio: Execução da Aplicação

6.3. Teste Complexo

Para o teste final o autor tentou fazer um teste mais complexo. É esperado para este teste uma aplicação que cumpra os seguintes requisitos:

- 4 tarefas;
- Uso de mensagens para troca de informações entre tarefas;
- Um contador para ativar a tarefa que envia mensagens em intervalos periódicos;
- Quando existir uma mensagem, ativar a tarefa que a recebe;
- Cada tarefa ter uma prioridade diferente;
- A tarefa de prioridade mais baixa ativa uma com uma prioridade mais alta;
- Imprimir no ecrã as mensagens trocadas;
- Usar as funções Hook para assinalar a Inicialização do Sistema e seu encerramento;

- Usar as funções "*Hook*" que serão executadas antes e depois das tarefas para saber o ID da tarefa que inicia execução e termina execução;

- Terminar o sistema quando o número enviado por mensagem for maior que quatro.

Para criar uma aplicação que cumpra todos os requisitos foi feito o desenho de sistema como se vê na Figura 75.

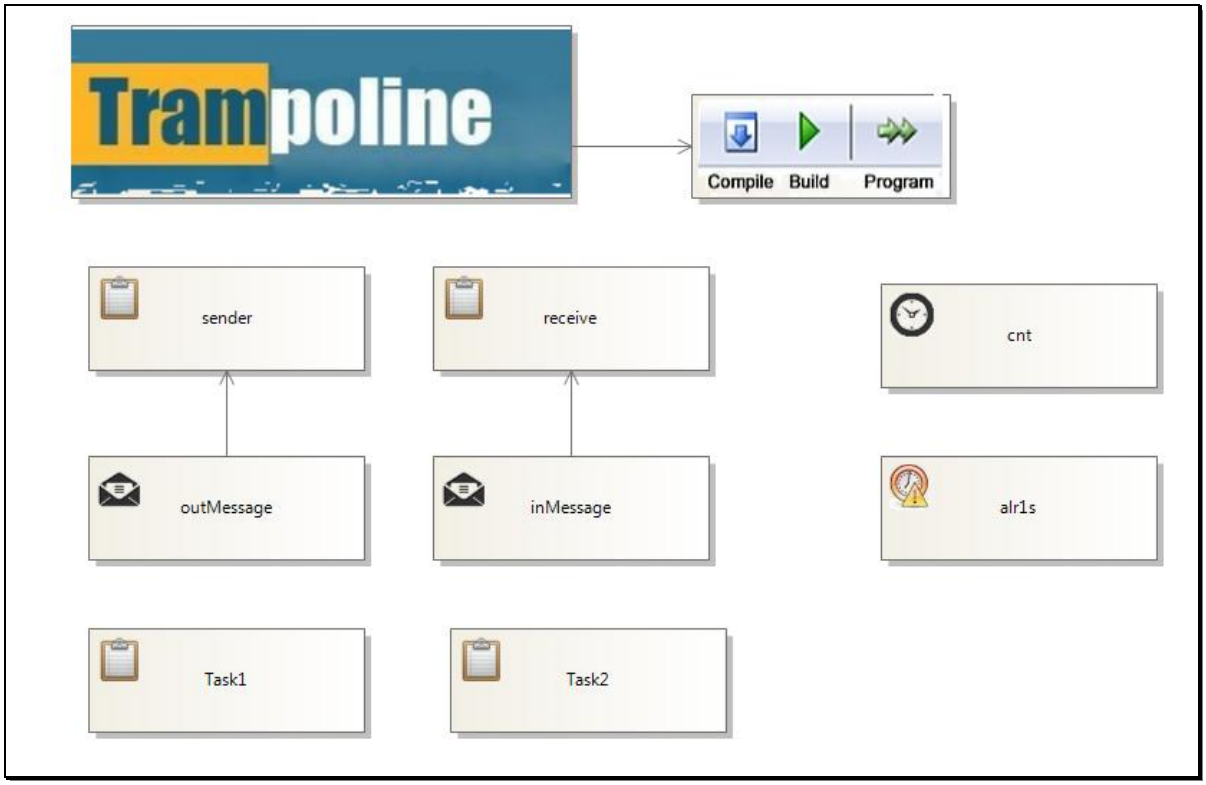


Figura 75 – Teste Complexo: Desenho do Sistema

Todos os componentes desenhados têm de ser devidamente configurados para corresponder aos requisitos propostos. A Figura 76 mostra como todos os componentes foram configurados e todos os seus valores.

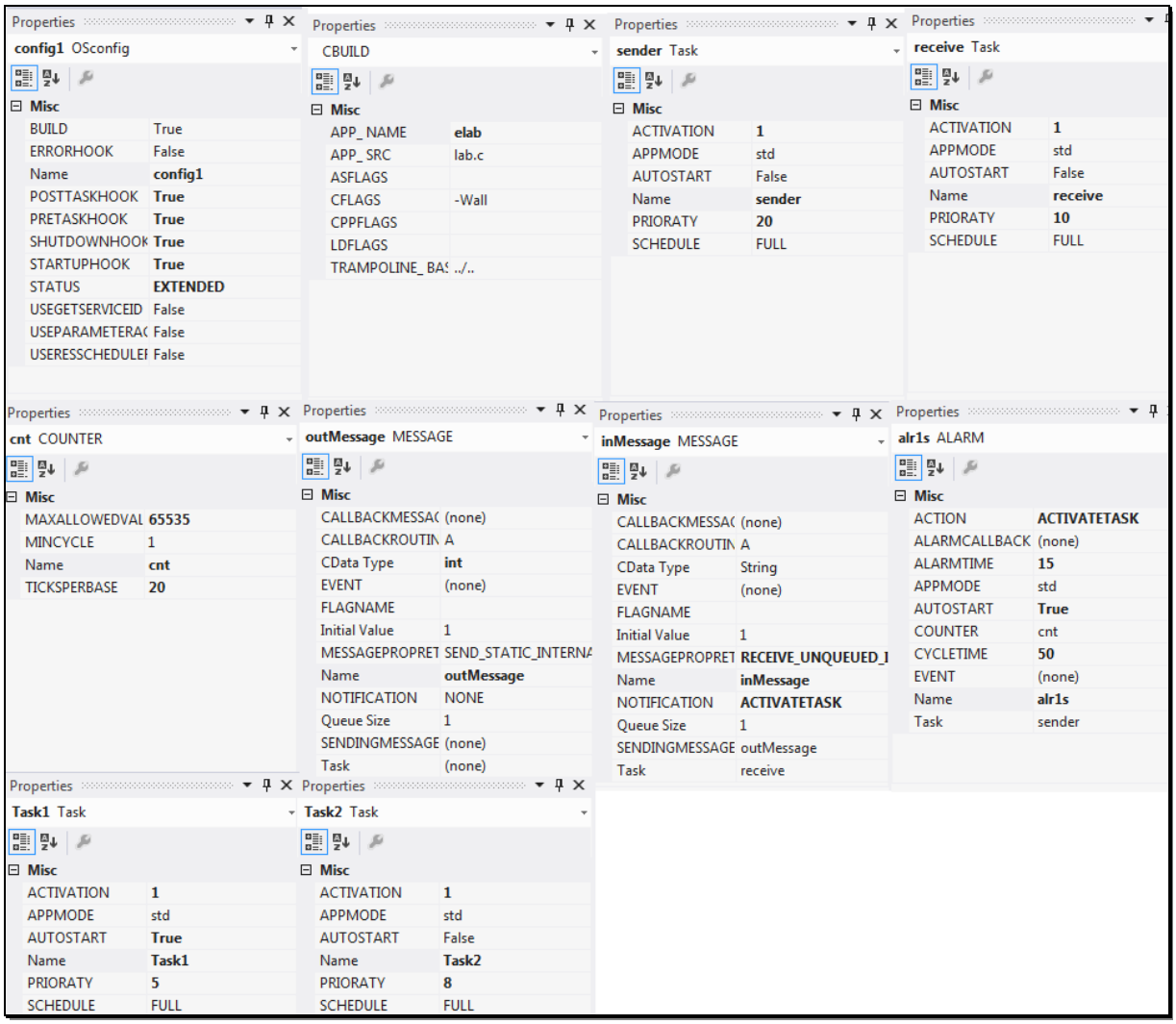


Figura 76 – Teste Complexo: Configuração das propriedades

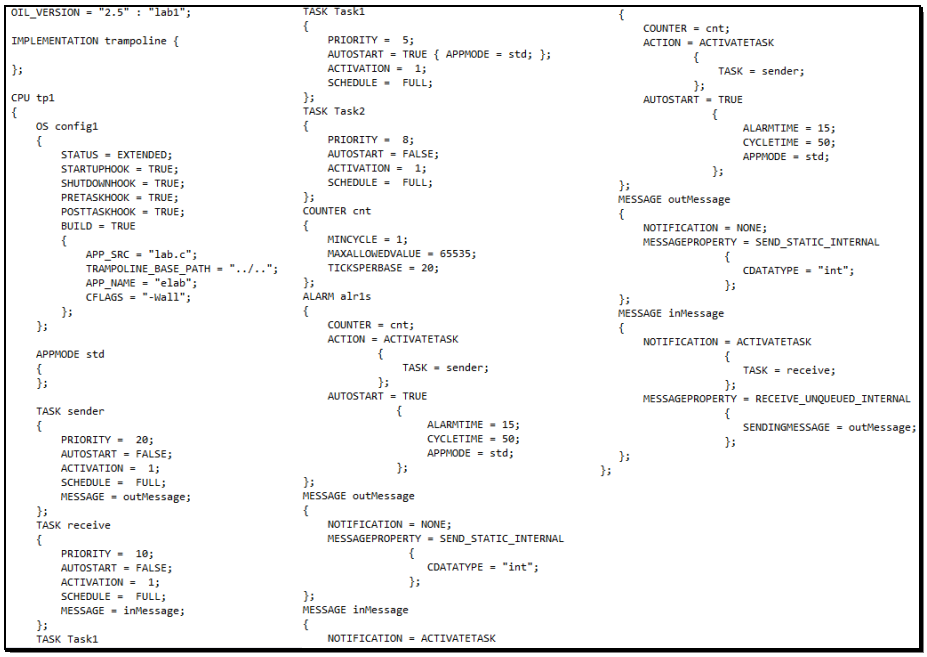
Depois de estar o sistema completamente desenhado e configurado para corresponder aos requisitos, passa-se à parte da transformação das "*templates*" e assim à criação dos ficheiros C, OIL e XML. A Figura 77 representa um pequeno excerto do ficheiro XML com o IP da aplicação desenhada.

```
<?xml version="1.0" encoding="UTF-8" ?>
<spirit:component>
    <spirit:vendor>Trampoline</spirit:vendor>
    <spirit:library>Trampoline</spirit:library>
    <spirit:name>tp1</spirit:name>
    <spirit:version>1.0</spirit:version>
</spirit:component>
<spirit:fileSets>
    <spirit:fileSet>
      <spirit:name>cSources</spirit:name>
      <spirit:file>
        <spirit:name>trunk/autosar/Os.h</spirit:name>
        <spirit:fileType>cSource</spirit:fileType>
        <spirit:isIncludeFile spirit:externalDeclarations="false">false</spirit:isIncludeFile>
      </spirit:file>
      <spirit:file>
        <spirit:name>trunk/autosar/tpl_as_action.c</spirit:name>
        <spirit:fileType>cSource</spirit:fileType>
        <spirit:isIncludeFile spirit:externalDeclarations="false">false</spirit:isIncludeFile>
      </spirit:file>
      <spirit:file>
        <spirit:name>trunk/autosar/tpl_as_action.h</spirit:name>
        <spirit:fileType>cSource</spirit:fileType>
        <spirit:isIncludeFile spirit:externalDeclarations="false">false</spirit:isIncludeFile>
      </spirit:file>
      <spirit:file>
        <spirit:name>trunk/autosar/tpl_as_app_kernel.c</spirit:name>
        <spirit:fileType>cSource</spirit:fileType>
        <spirit:isIncludeFile spirit:externalDeclarations="false">false</spirit:isIncludeFile>
      </spirit:file>
      <spirit:file>
        <spirit:name>trunk/autosar/tpl_as_app_kernel.h</spirit:name>
        <spirit:fileType>cSource</spirit:fileType>
        <spirit:isIncludeFile spirit:externalDeclarations="false">false</spirit:isIncludeFile>
      </spirit:file>
      <spirit:file>
        <spirit:name>trunk/autosar/tpl_as_application.c</spirit:name>
        <spirit:fileType>cSource</spirit:fileType>
        <spirit:isIncludeFile spirit:externalDeclarations="false">false</spirit:isIncludeFile>
      </spirit:file>
      <spirit:file>
        <spirit:name>trunk/autosar/tpl_as_application.h</spirit:name>
        <spirit:fileType>cSource</spirit:fileType>
        <spirit:isIncludeFile spirit:externalDeclarations="false">false</spirit:isIncludeFile>
      </spirit:file>
```

```
        <COUNTER:MINCYCLE>1</COUNTER:MINCYCLE>
        <COUNTER:MAXALLOWEDVALUE>65535</COUNTER:MAXALLOWEDVALUE>
        <COUNTER:TICKSPERBASE>10</COUNTER:TICKSPERBASE>
    </esrg:COUNTER>
    <esrg:ALARM>
        <ALARM:Name>alr1s</ALARM:Name>
        <ALARM:COUNTER>cnt</ALARM:COUNTER>
        <ALARM:ACTION>ACTIVATETASK</ALARM:ACTION>
        <ALARM:TASK>sender</ALARM:TASK>
        <ALARM:EVENT></ALARM:EVENT>
        <ALARM:ALARMCALLBACKNAME></ALARM:ALARMCALLBACKNAME>
        <ALARM:AUTOSTART>True</ALARM:AUTOSTART>
        <ALARM:ALARMTIME>5</ALARM:ALARMTIME>
        <ALARM:CYCLETIME>50</ALARM:CYCLETIME>
        <ALARM:APPMODE>std</ALARM:APPMODE>
    </esrg:ALARM>
    <esrg:MESSAGE>
        <MESSAGE:Name>outMessage</MESSAGE:Name>
        <MESSAGE:NOTIFICATION>NONE</MESSAGE:NOTIFICATION>
        <MESSAGE:TASK></MESSAGE:TASK>
        <MESSAGE:EVENT></MESSAGE:EVENT>
        <MESSAGE:CALLBACKROUTINENAME>A</MESSAGE:CALLBACKROUTINENAME>
        <MESSAGE:MESSAGE></MESSAGE:MESSAGE>
        <MESSAGE:FLAGNAME></MESSAGE:FLAGNAME>
        <MESSAGE:MESSAGEPROPRETY>SEND_STATIC_INTERNAL</MESSAGE:MESSAGEPROPRETY>
        <MESSAGE:CDATATYPE>int</MESSAGE:CDATATYPE>
        <MESSAGE:SENDINGMESSAGE> = </MESSAGE:SENDINGMESSAGE>
        <MESSAGE:INITIALVALUE>1</MESSAGE:INITIALVALUE>
        <MESSAGE:QUEUESIZE>1</MESSAGE:QUEUESIZE>
    </esrg:MESSAGE>
    <esrg:MESSAGE>
        <MESSAGE:Name>inMessage</MESSAGE:Name>
        <MESSAGE:NOTIFICATION>ACTIVATETASK</MESSAGE:NOTIFICATION>
        <MESSAGE:TASK>receive</MESSAGE:TASK>
        <MESSAGE:EVENT></MESSAGE:EVENT>
        <MESSAGE:CALLBACKROUTINENAME>A</MESSAGE:CALLBACKROUTINENAME>
        <MESSAGE:MESSAGE></MESSAGE:MESSAGE>
        <MESSAGE:FLAGNAME></MESSAGE:FLAGNAME>
        <MESSAGE:MESSAGEPROPRETY>RECEIVE_UNQUEUED_INTERNAL</MESSAGE:MESSAGEPROPRETY>
        <MESSAGE:CDATATYPE>String</MESSAGE:CDATATYPE>
        <MESSAGE:SENDINGMESSAGE> = outMessage</MESSAGE:SENDINGMESSAGE>
        <MESSAGE:INITIALVALUE>1</MESSAGE:INITIALVALUE>
        <MESSAGE:QUEUESIZE>1</MESSAGE:QUEUESIZE>
    </esrg:MESSAGE>
</spirit:vendorExtensions>
```

Figura 77 – Teste Complexo: Excerto do XML com o IP do sistema

O Código OIL também gerado encontra-se exposto na Figura 78.



```
OIL_VERSION = "2.5" : "lab1";

IMPLEMENTATION trampoline {

};

CPU tp1
{
    OS config1
    {
        STATUS = EXTENDED;
        STARTUPHOOK = TRUE;
        SHUTDOWNHOOK = TRUE;
        PRETASKHOOK = TRUE;
        POSTTASKHOOK = TRUE;
        BUILD = TRUE
        {
            APP_SRC = "lab.c";
            TRAMPOLINE_BASE_PATH = "../..";
            APP_NAME = "elab";
            CFLAGS = "-Wall";
        };
    };

    APPMODE std
    {
    };

    TASK sender
    {
        PRIORITY =  20;
        AUTOSTART = FALSE;
        ACTIVATION =  1;
        SCHEDULE =  FULL;
        MESSAGE = outMessage;
    };
    TASK receive
    {
        PRIORITY =  10;
        AUTOSTART = FALSE;
        ACTIVATION =  1;
        SCHEDULE =  FULL;
        MESSAGE = inMessage;
    };
    TASK Task1
    {
        PRIORITY =  5;
        AUTOSTART = TRUE { APPMODE = std; };
        ACTIVATION =  1;
        SCHEDULE =  FULL;
    };
    TASK Task2
    {
        PRIORITY =  8;
        AUTOSTART = FALSE;
        ACTIVATION =  1;
        SCHEDULE =  FULL;
    };
    COUNTER cnt
    {
        MINCYCLE = 1;
        MAXALLOWEDVALUE = 65535;
        TICKSPERBASE = 20;
    };
    ALARM alr1s
    {
        COUNTER = cnt;
        ACTION = ACTIVATETASK
                {
                    TASK = sender;
                };
        AUTOSTART = TRUE
                    {
                        ALARMTIME = 15;
                        CYCLETIME = 50;
                        APPMODE = std;
                    };
    };
    MESSAGE outMessage
    {
        NOTIFICATION = NONE;
        MESSAGEPROPERTY = SEND_STATIC_INTERNAL
                        {
                            CDATATYPE = "int";
                        };
    };
    MESSAGE inMessage
    {
        NOTIFICATION = ACTIVATETASK
                        {
                            TASK = receive;
                        };
        MESSAGEPROPERTY = RECEIVE_UNQUEUED_INTERNAL
                        {
                            SENDINGMESSAGE = outMessage;
                        };
    };
};
```

Figura 78 – Teste Complexo: Código OIL

O último ficheiro é o de código C (Figura 79), é o ficheiro em que apenas o seu esqueleto é gerado.

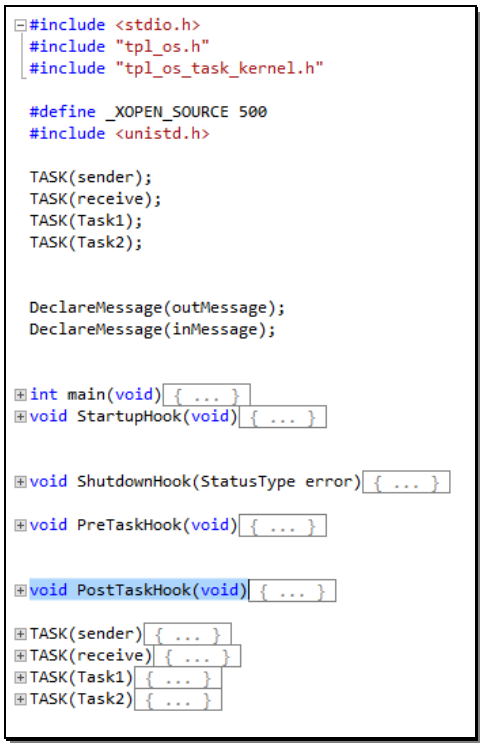


Figura 79 – Teste Complexo: Esqueleto do Código Gerado

É preciso então escrever o código de cada tarefa para que cumpra os requisitos propostos, mas também o código das funções "*Hook"* para que façam o que é previsto. A Figura 80 mostra o código que foi adicionado para criar a aplicação desejada.

```c
void StartupHook(void) {
    printf("System Start !\r\n");
}

void ShutdownHook(StatusType error) {
    printf("System Shutdown\r\n");
}

void PreTaskHook(void)
{
  TaskType id;
  GetTaskID(&id);
  printf("A Task que vai correr tem o ID: %d\r\n",id);
}

void PostTaskHook(void)
{
  TaskType id;
  GetTaskID(&id);
  printf("A Task que teminou tem o ID: %d\r\n",id);
}
```

```c
TASK(sender)
{
    static int value = 1;

    printf("Task Sender envia o Valor: %d\n",value);
    SendMessage(outMessage,&value);
    value++;
    TerminateTask();
}
TASK(receive)
{
    int value;

    ReceiveMessage(inMessage,&value);
    printf("Tasl Receive recebe o valor: %d\n",value);
    if (value > 4)
    {
        printf("Quinta Mensasagem %d\n",value);
        ShutdownOS(1);
    }
    TerminateTask();
}
TASK(Task1)
{
    unsigned int i;
    i = 0;
    while(1==1)
    {
    i++;
    printf("Task 1, activa Task 2.\n");

    ActivateTask(Task2);
    for(i=0;i<10E8; i++);
    }
}
TASK(Task2)
{
    printf("Task 2, Activada pela Task 1\n");
    TerminateTask();
}
```

Figura 80 – Teste Complexo: Possível Comportamento do Sistema

Depois de todo o código estar escrito e devidamente compilado, seguindo os passos explicados no primeiro teste, o programa está pronto a ser executado. A Figura 81 ilustra a execução do programa cumprindo todos os requisitos.

```
tese@ubuntu:~/trunk/labs/teste_f$ ./elab
System Start !
A Task que vai correr tem o ID: 0
Task 1, activa Task 2.
A Task que teminou tem o ID: 0
A Task que vai correr tem o ID: 1
Task 2, Activada pela Task 1
A Task que teminou tem o ID: 1
A Task que vai correr tem o ID: 0
A Task que teminou tem o ID: 0
A Task que vai correr tem o ID: 3
Task Sender envia o Valor: 1
A Task que teminou tem o ID: 3
A Task que vai correr tem o ID: 2
Tasl Receive recebe o valor: 1
A Task que teminou tem o ID: 2
A Task que vai correr tem o ID: 0
Task 1, activa Task 2.
A Task que teminou tem o ID: 0
A Task que vai correr tem o ID: 1
Task 2, Activada pela Task 1
A Task que teminou tem o ID: 1
A Task que vai correr tem o ID: 0
A Task que teminou tem o ID: 0
A Task que vai correr tem o ID: 3
Task Sender envia o Valor: 2
A Task que teminou tem o ID: 3
A Task que vai correr tem o ID: 2
Tasl Receive recebe o valor: 2
A Task que teminou tem o ID: 2
A Task que vai correr tem o ID: 0
Task 1, activa Task 2.
A Task que teminou tem o ID: 0
A Task que vai correr tem o ID: 1
Task 2, Activada pela Task 1
A Task que teminou tem o ID: 1
A Task que vai correr tem o ID: 0
A Task que teminou tem o ID: 0
A Task que vai correr tem o ID: 3
Task Sender envia o Valor: 3
A Task que teminou tem o ID: 3
A Task que vai correr tem o ID: 2
Tasl Receive recebe o valor: 3
A Task que teminou tem o ID: 2
A Task que vai correr tem o ID: 0
Task 1, activa Task 2.
A Task que teminou tem o ID: 0
A Task que vai correr tem o ID: 1
Task 2, Activada pela Task 1
A Task que teminou tem o ID: 1
A Task que vai correr tem o ID: 0
A Task que teminou tem o ID: 0
A Task que vai correr tem o ID: 3
Task Sender envia o Valor: 4
A Task que teminou tem o ID: 3
A Task que vai correr tem o ID: 2
Tasl Receive recebe o valor: 4
A Task que teminou tem o ID: 2
A Task que vai correr tem o ID: 0
Task 1, activa Task 2.
A Task que teminou tem o ID: 0
A Task que vai correr tem o ID: 1
Task 2, Activada pela Task 1
A Task que teminou tem o ID: 1
A Task que vai correr tem o ID: 0
A Task que teminou tem o ID: 0
A Task que vai correr tem o ID: 3
Task Sender envia o Valor: 5
A Task que teminou tem o ID: 3
A Task que vai correr tem o ID: 2
Tasl Receive recebe o valor: 5
Quinta Mensasagem 5
System Shutdown
tese@ubuntu:~/trunk/labs/teste_f$
```

Figura 81 – Teste Complexo: Execução da Aplicação

Capítulo 7

CONCLUSÃO

No sétimo e último capítulo, são apresentadas as conclusões tiradas desta dissertação com base no trabalho realizado. São também descritas propostas para um trabalho futuro e complementares a esta dissertação.

7.1. Conclusão

Despois de todo o trabalho desenvolvido e apresentado nos capítulos anteriores desta dissertação, o autor da mesma inferiu algumas conclusões, que serão agora apresentadas e explicadas do ponto de vista do autor.

Com a reutilização de código e a sua geração automática o tempo de produção diminui. Com a diminuição do tempo é seguro afirmar que se diminui também o custo de produção do mesmo, pois exige menos tempo de mão-de-obra. Com a redução do tempo de produção também existe um aumento na produtividade. Então, por consequência de isto tudo, o *Time-To-Market* do produto é reduzido dando ao produto e à empresa produtora uma vantagem sobre a concorrência [32].

A reutilização de código também traz vantagens ao nível dos testes de uma aplicação. Como o código a usar já foi testado e verificado, diminui a probabilidade de *bugs* [33].

Com um IDE que garante um maior nível de abstração e ao mesmo tempo permite gerar no final um sistema de mais baixo nível, é possível concluir-se que a utilização do mesmo facilita a geração de sistemas complexos.

A geração de um IP para o sistema operativo segundo o *standard* IP-XACT facilitará a incorporação deste IP em ambientes de desenvolvimento de SoC (*System-on-Chip*).

7.2. Trabalho Futuro

Com a realização de todos os objetivos propostos, é ainda possível estender a *framework* criada acrescentando-lhe funcionalidades para maior benefício do utilizador. Para trabalho futuro nesta dissertação serão expostas quatro propostas diferentes.

1. Incorporar o compilador GOIL e C no IDE. Permitir que o IDE aceda diretamente ao compilador GOIL e C, e assim em vez de gerar para o utilizador os ficheiros C e OIL, gerar apenas o ficheiro executável pronto a executar, evitando assim o uso ferramentas externas.

2. Fornecer suporte para um maior número de placas de desenvolvimento. Com a integração do compilador C no IDE, seria depois aconselhável incorporar os compiladores para placas de desenvolvimento para o qual o sistema operativo já tem *porting* disponível e assim permitir a compilação do sistema operativo para vários ambientes diferentes.

3. Permitir a geração do sistema operativo a partir do seu IP IP-XACT. Acrescentar ao IDE a funcionalidade de fazer o trabalho inverso, permitir que quando o utilizador tem o IP em XML que descreve um sistema operativo, possa gerar o ficheiros finais ou o executável do sistema.

4. Estender para permitir a modelação de *hardware*. Permitir que o IDE também faça em paralelo com a modelação do *software* a do *hardware* onde o sistema irá ser executado.

REFERENCIAS BIBLIOGRAFICAS

[1] A. Hergenhan and G. Heiser, “Operating Systems Technology for Converged ECUs.”

[2] AUTOSAR, “AUTOSAR.ORG.” [Online]. Available: http://www.autosar.org/index.php?p=1&up=2&uup=3&uuup=3&uuuup=0. [Accessed: 23-Sep-2013].

[3] “What is integrated development environment (IDE)? - Definition from WhatIs.com.” [Online]. Available: http://searchsoftwarequality.techtarget.com/definition/integrated-development-environment. [Accessed: 01-Sep-2014].

[4] “version control - Multiple Programmers in Software Development. How do we work on the same code and ensure it is always updated? - Stack Overflow.” [Online]. Available: http://stackoverflow.com/questions/1447653/multiple-programmers-in-software-development-how-do-we-work-on-the-same-code-an. [Accessed: 01-Sep-2014].

[5] “Five Integrated Development Environment applications - TechRepublic.” [Online]. Available: http://www.techrepublic.com/blog/five-apps/five-integrated-development-environment-applications/. [Accessed: 01-Sep-2014].

[6] “Integrated development environment.” [Online]. Available: https://www.princeton.edu/~achaney/tmve/wiki100k/docs/Integrated_development_environment.html. [Accessed: 01-Sep-2014].

[7] “The importance of a good IDE | Remus Stratulat - On the Stre@m.” [Online]. Available: http://www.stratulat.com/blog/the-importance-of-a-good-ide. [Accessed: 01-Sep-2014].

[8] “Qt - IDE & Tools.” [Online]. Available: http://qt.digia.com/product/Qt-Framework/IDE--Tools/. [Accessed: 06-Sep-2014].

[9] “Qt 4.7.0: Supported Platforms.” [Online]. Available: http://doc.qt.digia.com/qt-maemo/supported-platforms.html. [Accessed: 06-Sep-2014].

[10] “QtCreatorWhitepaper | Qt Wiki | Qt Project.” [Online]. Available: http://qt-project.org/wiki/QtCreatorWhitepaper. [Accessed: 06-Sep-2014].

[11] “Module concept.” [Online]. Available: http://docs.huihoo.com/qt/qtextended/4.4/selectingmodules.html. [Accessed: 06-Sep-2014].

[12] F. B. Faria, P. S. N. Lima, L. G. Dias, A. A. Silva, M. P. Costa, and T. J. Bittar, “Evolução e Principais Características do IDE Eclipse,” 2010.

[13] “Posts about real advantages of using Eclipse on Altabel Group’s Blog.” [Online]. Available: https://altabel.wordpress.com/tag/real-advantages-of-using-eclipse/. [Accessed: 06-Sep-2014].

[14] G. C. Murphy, M. Kersten, L. Findlater, and B. Columbia, “How Are Java Software Developers Using the Eclipse IDE?,” 2006.

[15] “Getting Started with the .NET Framework.” [Online]. Available: http://msdn.microsoft.com/en-us/library/hh425099.aspx. [Accessed: 08-Sep-2014].

[16] “Overview of the .NET Framework.” [Online]. Available: http://msdn.microsoft.com/en-us/library/zw4w595w.aspx. [Accessed: 08-Sep-2014].

[17] M. De Cock, “Comparison of Visual Studio’s VMSDK and AToM3.”

[18] G. A, “AUTOSAR Layered Software Architecture.” [Online]. Available: http://www.autosar.org/download/AUTOSAR_LayeredSoftwareArchitecture.pdf. [Accessed: 25-Oct-2013].

[19] Autosar, “AUTOSAR Layered Software Architecture.” [Online]. Available: http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/general/auxiliary/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf. [Accessed: 27-Nov-2014].

[20] Y. Trinquet and N. C. France, “Trampoline An OpenSource Implementation of the OSEK / VDX RTOS Specification,” *IEEE Conf. Emerg. Technol. Fact. Autom.*, pp. 62–69, 2006.

[21] “OSEK/VDX Goals and Motivation.” [Online]. Available: http://portal.osek-vdx.org/index.php?option=com_content&task=view&id=4&Itemid=4. [Accessed: 19-Aug-2014].

[22] Mb. Jochem Spohr, “OSEK/VDX Operating System,” 2005.

[23] “Trampoline - Open Source RTOS Project.” [Online]. Available: http://trampoline.rts-software.org/spip.php?article1. [Accessed: 19-Aug-2014].

[24] “IP-XACT - Accellera Systems Initiative.” [Online]. Available: http://www.accellera.org/activities/committees/ip-xact/. [Accessed: 28-Aug-2014].

[25] J. A. Swanson, “Building an IP-XACT Design and Verification Environment with DesignWare IP.” [Online]. Available: http://www.synopsys.com/Company/Publications/DWTB/Pages/dwtb-IP-XACT-design-jun2012.aspx. [Accessed: 28-Aug-2014].

[26] “IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tools Flows,” *IEEE Std 1685-2009*, pp. C1–360, 2010.

[27] E. Murphy-Hill, C. Parnin, and A. P. Black, “How we refactor, and how we know it,” *2009 IEEE 31st Int. Conf. Softw. Eng.*, pp. 287–297, 2009.

[28] H. Liu, Y. Gao, and Z. Niu, “An Initial Study on Refactoring Tactics,” *2012 IEEE 36th Annu. Comput. Softw. Appl. Conf.*, pp. 213–218, Jul. 2012.

[29] M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ru, and C. Valot, “SOS : An Object-Oriented Operating System | Assessment and Perspectives 1 Introduction,” vol. 2, no. 4, 1991.

[30] OS working group, “OSEK Implementation Language Specification 2.3,” 2001.

[31] Q. M. NGUYEN, “Planning in Software Project Management - An Empirical Research of Software Companies in Vietnam,” University of Fribourg, Switzerland, 2006.

[32] “Improving Product Time to Market (TTM) | Arena Solutions.” [Online]. Available: http://www.arenasolutions.com/resources/articles/time-to-market. [Accessed: 29-Jul-2014].

[33] “Code reuse - DocForge.” [Online]. Available: http://docforge.com/wiki/Code_reuse. [Accessed: 29-Jul-2014].

[34] “Douglas W. Jones’s punched card index.” [Online]. Available: http://homepage.cs.uiowa.edu/~jones/cards/history.html. [Accessed: 05-Sep-2014].

[35] D. W. Jones, “History of the punch card,” *Computing fundamentals*. [Online]. Available: http://whatis.techtarget.com/reference/History-of-the-punch-card. [Accessed: 05-Sep-2014].

[36] M. Chalabine, “Integrated Development Environments (IDEs) Eclipse.” [Online]. Available: http://www.ida.liu.se/~chrke/courses/SWE/IDEs_Eclipse.pdf. [Accessed: 05-Sep-2014].

[37] “Maestro I.” [Online]. Available: http://www.in.com/maestro-i/biography-180639.html. [Accessed: 05-Sep-2014].

[38] “The History of Visual Development Environments.” [Online]. Available: http://www.mendix.com/think-tank/the-history-of-visual-development-environments-imagine-theres-no-ides-its-difficult-if-you-try/. [Accessed: 05-Sep-2014].

[39] “Visual Studio Online User Plans.” [Online]. Available: http://www.visualstudio.com/products/visual-studio-online-user-plans-vs. [Accessed: 05-Sep-2014].

Anexo A.

Evolução

O uso de IDEs só se tornou possível quando se começou a desenvolver programas através de consolas ou terminais de computadores. No início eram usadas cartas furadas, que podiam ser de papel, cartão, ou outros materiais mais resistentes como alumínio, para criar programas e guardar dados.

No princípio, as cartas furadas (Figura 1) eram usadas como forma de guardar informação estatística [34]. Foi mais tarde quando Charles Babbage viu que poderiam ser usados para controlar os cálculos da sua proposta para uma máquina analítica.

Figura 82 – Jacquard Cards

Mesmo Baggage nunca tendo construído a sua máquina, a proposta que lançou serviu para prevenir que as empresas que se seguiram no uso de cartões não pudessem reclamar direitos ou patentes da ideia[34].

Mais tarde, Hollerth apareceu com os seus cartões, que eram manuseados como baralhos de cartas de jogo [34] (Figura 83).

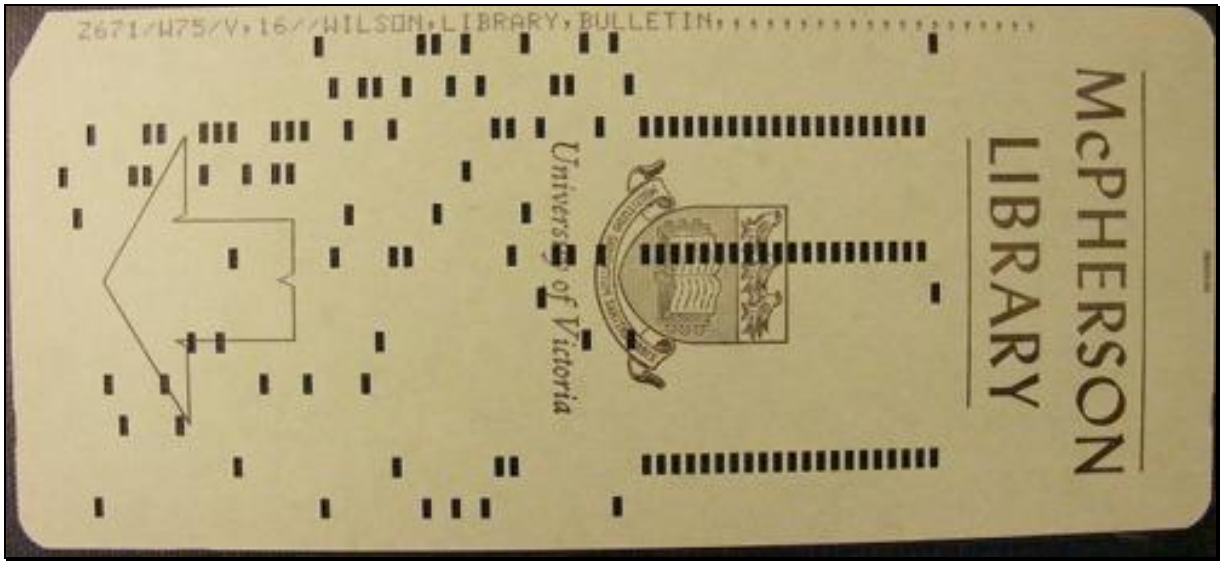


Figura 83 – Punched Card

O principal contributo de Hollerith não foi em guardar dados através de buracos em cartões, mas no desenvolvimento de máquinas para processar os dados e sistemas para processar esses dados [35].

Mais tarde os cartões foram adaptados com formatos específicos para as necessidades dos programadores.

A primeira linguagem com a ideia de um IDE foi Darthmout BASIC [36]. Foi a primeira linguagem a vir com o conceito de IDE.

O primeiro IDE, para *software*, a sair para o mercado foi o Maestro I, um produto da Softlab Munich [37].

Mas com o aparecimento de sistemas operativos para computadores com ambientes gráficos e noções de janelas, como é o caso do Windows, os IDEs foram-se alterando e ficando mais complexos e completos, mas ao mesmo tempo mais simples de usar. O uso de IDEs com interface gráfica fez disparar o desenvolvimento de *software* de muitas empresas, como é o caso da empresa Delphi [38].

Mais recentemente apareceram os IDEs com caracteríscas próprias para trabalho em equipa e que permitem incorporar *plugins*. Permitem que o projeto a ser desenvolvido esteja guardado em *Cloud* e que seja editado por mais que uma pessoa [39].