Universidade do Minho
Escola de Engenharia

Rui Pedro Oliveira Machado

Desenvolvimento de uma framework compatível com IP-XACT para a criação e reutilização de sistemas de hardware e software

Novembro de 2014

UMinho | 2014
Rui Pedro Oliveira Machado
Desenvolvimento de uma framework compatível com IP-XACT
para a criação e reutilização de sistemas de hardware e software

Universidade do Minho
Escola de Engenharia

Rui Pedro Oliveira Machado

Desenvolvimento de uma framework compatível com IP-XACT para a criação e reutilização de sistemas de hardware e software

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

Trabalho efetuado sob a orientação do
Professor Doutor Adriano Tavares

Novembro de 2014

DECLARAÇÃO

Nome: Rui Pedro Oliveira Machado

Correio electrónico: rhorigamy@gmail.com

Tlm.: 914094895

Número do Bilhete de Identidade: 13737459

Título da dissertação:

Desenvolvimento de uma Framework compatível com IP-XACT

para criação e reutilização de sistemas de Hardware e Software

Ano de conclusão: 2014

Orientador:

Adriano José da Conceição Tavares

Designação do Mestrado:

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

Área de Especialização: Sistemas Embebidos

Escola de Engenharia

Departamento de Electrónica Industrial

Guimarães, ____/____/_______

Assinatura: ____

“Si se cree y se trabaja, se puede!” – Diego Simeone, 2014

AGRADECIMENTOS

Em primeiro lugar gostaria de agradecer aos meus pais, José Augusto Machado e Maria Laurentina Oliveira, por todo o apoio prestado ao longo dos cinco anos do Mestrado, sem o qual esta dissertação e consequente finalizar do meu percurso académico não seriam possíveis. A eles o meu muito obrigado pelo apoio financeiro e psicológico e por terem sempre as palavras certas no momento certo.

Ao meu orientador Professor Doutor Adriano Tavares pelo apoio prestado, confiança depositada e conhecimento transmitido ao longo do desenvolvimento desta dissertação. Um obrigado especial ao professor Doutor Jorge Cabral pelas aulas de Microprocessadores I que me levaram a optar pela especialização de Sistemas Embebidos

Ao ESRG (*Embedded Systems Research Group*) do departamento de Eletrónica Industrial da Universidade do Minho por proporcionar todas as condições necessárias para o desenvolvimento da presente dissertação de mestrado, em especial para o Professor Doutor Nuno Cardoso pela disponibilidade demonstrada e tempo despendido sempre que precisei.

Aos colegas que me acompanharam ao longo do curso, em especial aos meus colegas de laboratório Pedro Matos, Tiago Vasconcelos, João Martins, Eurico Moreira, Davide Guimarães, Filipe Alves e Vasco Lima pela entreajuda, companheirismo e momentos de descontração. Queria agradecer ainda ao meu tio e colega de curso Jorge Fernandes, o Avô Cantigas, por me aconselhar a optar pelo curso de Engenharia Eletrónica.

Aos meus amigos, Antero Freitas e Marta Pereira por toda a confiança transmitida, apoio prestado e por estarem presentes nos momentos memoráveis da minha vida.

Por fim, mas não menos importante, à minha namorada Juliana Martins por me conseguir aturar nos momentos de maior stress, por me conseguir animar, ou pelo menos tentar, nos momentos de maior desilusão e por todo o mimo e força depositados em mim.

RESUMO

Com a crescente competitividade do mercado atual, existe a necessidade de chegar o mais rápido possível ao mercado, na tentativa de ganhar vantagem sobre a concorrência [1], [2]. No entanto, a crescente complexidade dos sistemas atuais aumenta o tempo gasto nos processos de verificação e validação, fazendo com que o tempo de desenvolvimento de um protótipo aumente, resultando num consequente aumento do *time to market* [1], [3], [4]. O tempo despendido em esforços repetidos de engenharia é também um fator a combater, por forma a maximizar o desempenho e minimizar o capital investido. Consequentemente, temas como a automatização dos processos de desenvolvimento e teste de sistemas e a reutilização de sistemas já desenvolvidos têm ganho relevância.

A presente dissertação insere-se neste contexto na medida em que visa o desenvolvimento de uma *framework* compatível com IP-XACT que cubra as diversas fases envolvidas no *design flow* de sistemas baseados em FPGA. Uma *framework* é um conjunto de ferramentas que auxiliam o desenvolvimento de sistemas dentro de um determinado domínio de aplicações, e tem como principais características ser fácil de usar, eficiente, extensível e seguro. O IP-XACT é um *standard* desenvolvido pelo *Spirit Group*, concebido em torno do conceito de reutilização de IPs (*Intelectual Property*) [5]. Em 2009 o IEEE lançou o *standard* 1685 que descreve o IP-XACT [6], [7]. Com a sua base em XML, o *standard* oferece um mecanismo de descrição de IP, independente da linguagem de implementação do componente [2], [5], [6].

Assim, a *framework* proposta pretende possibilitar a criação, gestão e reutilização de IPs baseados no *standard* IP-XACT, efetuando validações e verificações em tempo de desenvolvimento, com o objetivo de diminuir o número de erros cometidos no processo de *design* permitindo, ao mesmo tempo, a geração completa do sistema e de todos os *batchs* necessários para o correto *deployment* na respetiva plataforma, baseando-se para o efeito numa abordagem generativa (*Generative Programming*).

Palavras-chave: IP-XACT, *Framework*, XML, XSLT, *Intellectual Property* (IP).

ABSTRACT

The huge competition in the actual market requires special focus on to the TTM (Time-To-Market) pressure in order to gain advantage over the competitors. The growing complexity of the actual systems enlarges not only the time-to-prototype and the time-to-market but it also requires an efficient time effort around repetitive engineering tasks in order to maximize the efficiency and minimize the money investment. In this line of thought, essays about system development and test automation and system reutilization have been gaining relevance.

The current thesis appears in this context, because it aims the development of an IP-XACT enabled *framework* that encompasses the different phases in a FPGA system *design flow*. A *framework* is an ecosystem of tools that assists the system development process around an application family in a specific domain and has, as its main characteristics, easy to use, efficiency, extensibility, safety and security. IP-XACT is a *standard* developed by the spirit group around the concept of IP reutilization. In 2009, IEEE develop the 1685 *standard* that describes the IP-XACT. Based on XML, the *standard* offers a mechanism to describe IP (Intellectual Properties) regardless its implementation programming language.

This way, the proposal *framework* intend to allow the IP creation and reutilization based on IP-XACT *standard*, with integrated design time verification and validation, in order to reduce the number of errors involved in the *design* process. At the same time, it allows the complete system generation, based on a generative programming (GP) approach, as well as all the *batchs* needed to deploy the system in the specific target.

Keywords: IP-XACT, *Framework*, XML, XSLT, Intellectual Property (IP).

ÍNDICE GERAL

ABREVIATURAS E SIGLAS

ASIC – Application Specific Integrated Circuit

CLI – Command Line Interface

EDA – Electronic *Design* Automation

ESL – Electronic System Level

FPGA – Field Programmable Gate Array

GP – Generative Programming

GUI – Graphical User Interface

HDL – *Hardware* Description Language

IDE – Integrated Development Environment

IEEE – Institute of Electrical and Electronics Engineers

IP – Intellectual Property

ISE – Integrated System Environment

NCD – Native Circuit Description

NGD – Native Generic Database

PCF – Physical Constraints File

RTL – Register Transfer Level

TCL – Tool Command Language

TGI – Tight Generator Interface

UCF – User Constraints File

VHDL – Very High Speed Integrated Circuits *Hardware* Description Language

XML – eXtensible Markup Language

XPath – eXtensible Markup Language Path Language

XST – Xilinx Synthesis Technology

XSLT – eXtensible Stylesheet Language Transformation

ÍNDICE DE FIGURAS

ÍNDICE DE TABELAS

Capítulo 1

INTRODUÇÃO

No presente capítulo será realizada uma breve contextualização do tema da dissertação. De seguida serão apresentados os objetivos da dissertação, terminando com uma descrição da organização da mesma.

1.1. Contextualização

A partir do momento em que uma empresa é criada, existem fatores com os quais esta tem de lidar, que não estão diretamente relacionados com a atividade para a qual a empresa foi idealizada. Um exemplo destes fatores é a competitividade com a concorrência, que implica a busca incessante pela maior quota possível de mercado. A fim de se afirmar, uma empresa tem a necessidade de garantir novos produtos, com um preço mais competitivo e eficiência superior. Perante a crescente competitividade do mercado atual, surge a necessidade de chegar o mais rápido possível ao mercado, na tentativa de ganhar vantagem sobre a concorrência [1], [2]. No entanto, o aumento gradual da complexidade dos sistemas atuais aumenta o tempo gasto nos processos de verificação e validação dos mesmos, fazendo com que o tempo necessário ao desenvolvimento de um protótipo aumente, o que pode resultar num aumento do *time to market* [1], [3], [4].

A utilização de ferramentas [1], como o Visual Studio da Microsoft ou o eclipse, favorecem os processos de desenvolvimento, dando suporte aos programadores nas tarefas de depuração. Estas *frameworks* oferecem ainda um conjunto de funcionalidades, como a coloração de palavras-chave, completação de texto e tabulação automática, que permitem um desenvolvimento de código mais organizado, facilitando a sua leitura e escrita.

A progressiva complexidade do *software* levou à criação de bibliotecas contendo código reutilizável, contribuindo para o agilizar de tarefas do programador e evitando que repetições sucessivas do mesmo código fossem necessárias. Com o aparecimento das FPGA (*Field Programmable Gate Array*), passou a ser possível programar *hardware*. Métodos como o desenvolvimento de bibliotecas de *hardware* foram aplicados não só para evitar repetições mas também para reaproveitar o trabalho já desenvolvido. No entanto continuava a existir um problema persistente em ambos os casos. A reutilização só é possível dentro do mesmo ambiente de desenvolvimento, isto é, todo o código criado, quer em *software*, quer em *hardware*, é dependente da plataforma.

No sentido de possibilitar uma reutilização, independente da plataforma alvo, foi criado um *standard* com o nome de IP-XACT. Desta forma é melhorada a interoperabilidade de IPs e ferramentas, e a reutilização dos módulos de IP, permitindo às empresas dividir os custos e riscos do desenvolvimento, evitando assim o aumento do esforço de produção [4].

A utilização de um *standard* como o IP-XACT não se justifica se o programador for responsável por gerir essas as bibliotecas (que podem chegar às centenas de IPs muito facilmente). De modo a possibilitar o seu uso de forma prática, é necessário a existência de uma *framework* compatível com IP-XACT, isto é, que cumpre o *standard* IP-XACT em todos os passos do processo de desenvolvimento de um projeto. Mais uma vez, apesar de ser fornecida uma forma cómoda, esta solução obriga à utilização de uma *framework* específica, havendo apenas portabilidade entre *frameworks* compatíveis com IP-XACT.

Atualmente o *software* Kactus2, desenvolvido pela Tampere University of Technology, permite a gestão de um repositório de IPs que seguem o *standard*. Contudo o *software* apresenta algumas limitações uma vez que não permite a simulação dos IPs desenvolvidos. A Magillem possui também um *software* compatível com o IP-XACT disponibilizado sob a forma de um *plug-in* para o Eclipse, que dá suporte a todas as versões do *standard*.

Desta forma, propõe-se o desenvolvimento de uma *framework* que faça a gestão de um repositório de IPs de acordo com o *standard*, que consiga criar IPs partindo de ficheiros Verilog e VHDL, e que seja capaz de gerar os ficheiros IP-XACT dos IPs. Pretende-se também que a *framework* possibilite a geração dos ficheiros Verilog e VHDL para os IPs criados de raiz.

1.2. Objetivos

O objetivo primário desta dissertação enquadra-se no desenvolvimento de uma *framework* compatível com o *standard* IEEE 1685 IP-XACT. Para a sua concretização foram identificados objetivos de menor granularidade que serviram como *milestones* ao longo do desenvolvimento da presente dissertação, os quais passo a referir.

O primeiro objetivo passa por criar um gestor de um repositório IP-XACT. Este gestor deve permitir a leitura, criação e reconfiguração de todo o tipo de IPs que se encontrem no formato IP-XACT. Para que este objetivo se dê como concluído é necessário um estudo exaustivo do *standard* em causa para posterior criação de um diagrama de classes, que constituirá a base do gestor do repositório.

O segundo objetivo é a construção de um GUI (*Graphical User Interface*) para a conceção de IPs a partir de IPs já existentes no repositório. Para que o ambiente gráfico seja o mais intuitivo possível, um estudo sobre a interação entre o utilizador e as ferramentas de desenvolvimento deve ser realizado. O segundo objetivo será concluído aquando da criação e incorporação de um gerador de código no GUI, que permitirá a geração dos ficheiros Verilog que representam a estrutura do IP desenvolvido.

Finalmente, o terceiro objetivo passa por implementar mecanismos de invocação de ferramentas externas. Estes mecanismos devem ser o mais transparentes possíveis para o utilizador, devendo as ferramentas externas, necessárias a cada IP, serem definidas no momento da criação de um novo IP para o repositório e não no momento da compilação do IP. Este objetivo possibilita alargar o âmbito de atuação da *framework* a desenvolver, evitando a necessidade de utilizar mais que uma ferramenta ao longo do processo de desenvolvimento de um IP.

1.3. Organização da Dissertação

A presente dissertação encontra-se dividida em seis capítulos, nomeadamente: introdução; estado de arte; especificações do sistema; implementação do sistema; testes e resultados e conclusão. Seguidamente será apresentado um pequeno resumo sobre o conteúdo de cada um dos capítulos referidos.

O segundo capítulo retrata o estado atual das tecnologias utilizadas no desenvolvimento desta dissertação. Começa-se com uma explicação sobre o *standard* em questão, o IP-XACT, seguido de uma descrição comparativa entre o *design flow* de sistemas em ASIC e em FPGA. Explica-se o que é uma *framework* e que funcionalidades devem estar presentes neste tipo de ferramentas, sendo referidas as vantagens e desvantagens da sua utilização. Por fim, o capítulo termina com uma pequena conclusão onde é identificada o ambiente integrado de desenvolvimento a utilizar para a criação da *framework*.

No capítulo 3 são abordadas todas as especificações do sistema a implementar. O capítulo é iniciado com uma descrição do *standard* IP-XACT seguido de uma divisão do sistema a implementar em três módulos fundamentais, (i) o gestor de repositório, (ii) o GUI, (iii) e o gerador de código e mecanismos de invocação de ferramentas externas. Após apresentados os motivos para esta divisão proceder-se-á a uma análise de cada um dos módulos individualmente. O capítulo termina com uma abordagem pormenorizada do ambiente e plataformas de desenvolvimento que serão utilizadas na implementação do sistema.

O quarto capítulo apresenta a implementação do sistema proposto. São apresentados todos os passos realizados no processo de desenvolvimento da *framework*, divididos de acordo com os três módulos identificados no capítulo anterior. De um modo geral, este capítulo descreve todo o trabalho realizado no decorrer da presente dissertação, desde a criação do gestor do repositório, à validação dos IPs criados, passando pelo processo de implementação da interface gráfica, até ao desenvolvimento e geração de código e invocação dos compiladores externos à aplicação.

No capítulo 5 são apresentados os testes realizados e os resultados experimentais obtidos. A fim de validar o sistema implementado foram realizados testes aos IPs e ao código gerado automaticamente. A verificação dos IPs foi realizada recorrendo ao *software* Kactus2 que é uma ferramenta que respeita o *standard* utilizado. Para a verificação do código gerado automaticamente recorreu-se à placa de desenvolvimento Basys2 da Xilinx e à plataforma de desenvolvimento Cyclone2 DE-70 da Altera.

Algumas conclusões relativamente ao trabalho realizado são apresentadas no capítulo 6. Para além das principais conclusões são ainda sugeridas algumas propostas de trabalho futuro, visando melhorar e aumentar as funcionalidades da *framework* desenvolvida.

O documento termina com a apresentação dos Anexos referenciados ao longo do presente documento.

Capítulo 2

ESTADO DE ARTE

O presente capítulo inicia-se com uma breve descrição do *standard* do IEEE 1685 IP-XACT, ao qual a *framework* da presente dissertação deve oferecer suporte. De seguida é realizada uma introdução ao *design flow*, sendo abordados com maior detalhe o *design flow* de um ASIC e de uma FPGA. Por forma a analisar o panorama atual, são apresentados alguns exemplos de ferramentas que oferecem suporte ao *design flow* e ao IP-XACT em simultâneo.

É apresentada a definição de *framework* e explicada a importância da sua utilização em projetos de desenvolvimento, seguida de uma evolução das mesmas ao longo do tempo.

O capítulo termina com uma breve conclusão onde é definido o Ambiente Integrado de Desenvolvimento de *software* a utilizar para a conceção da *framework* proposta.

2.1. IP-XACT

O IP-XACT é um *standard* desenvolvido pelo *Spirit Group*, criado com base no conceito de reutilização de IPs [5]. Em 2009 o IEEE lançou o *standard* 1685 que descreve o IP-XACT [6], [7]. Com a sua base em XML, o *standard* oferece um mecanismo de descrição de IPs, independente da linguagem de implementação do componente [2], [5], [6]. Ou seja, o *standard* não possui qualquer tipo de informação relativa à implementação dos componentes de *hardware*, em vez disso, fornece informações relativas às interfaces do sistema, ao mapeamento da memória, aos compiladores e aos ficheiros externos (estes podem, ou não, conter informações relativas à implementação do IP). Na Figura 1 é apresentado um dos sete principais elementos do *standard*.

Através da utilização deste *standard* assegura-se a compatibilidade entre fornecedores de IP e permite-se a troca de bibliotecas entre plataformas de desenvolvimento [5]. Desta forma não é necessário despender tempo a desenvolver algo que já foi implementado por outra entidade, permitindo que os projetistas se concentrem na implementação das funcionalidades do novo IP.

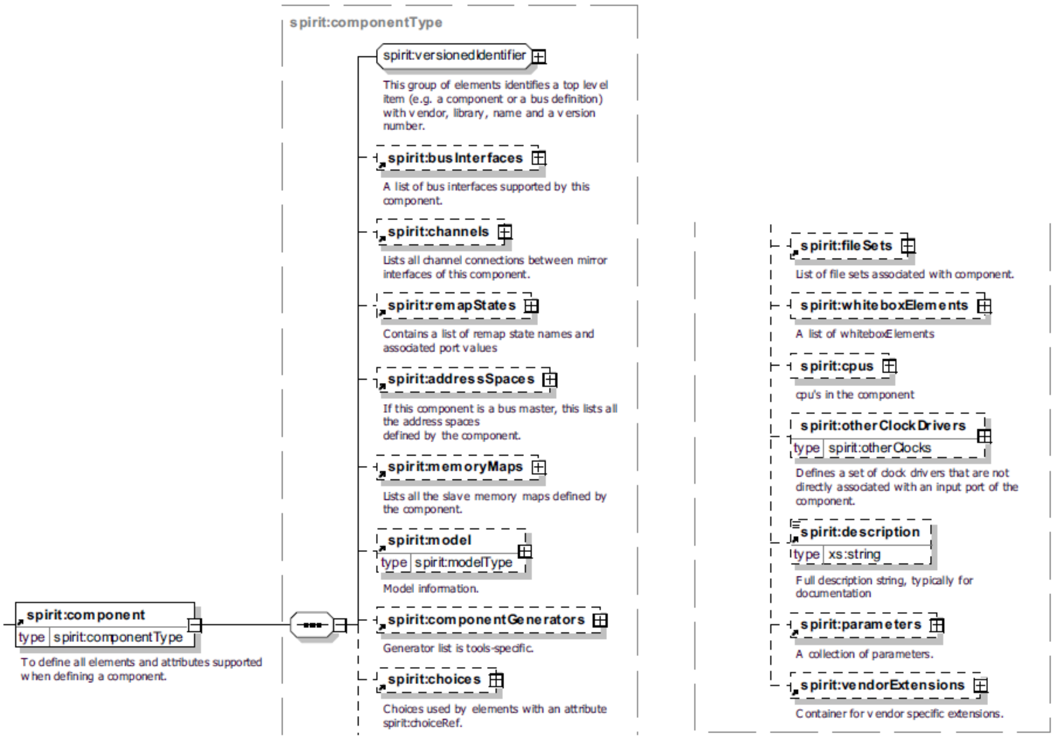


Figura 1 – XML Schema do elemento IP-XACT utilizado para descrever um componente [5]

Para que uma *framework* seja considerado compatível com IP-XACT é necessário que forneça suporte para a importação/exportação de IPs e sistemas, no formato IP-XACT, e utilizar a TGI (*Tight Generator Interface*) para realizar a interface com geradores externos. Deste modo, o *standard* não restringe de modo algum, a forma como deve ser efetuado o processo de leitura/escrita, assim como também não impõe restrições relativas à forma de armazenamento da informação na *framework* [5].

Atualmente já existem *frameworks* para gestão de IPs no formato IP-XACT. Uma dessas *frameworks* é o Kactus2, desenvolvido graças ao projeto *funbase-project*, da Tampere University of Technology. O Kactus2 será explorado numa das secções seguintes, sendo as suas características evidenciadas e analisadas.

2.2. *Design Flow*

Com o aumento da complexidade dos sistemas de *hardware*, tornou-se fundamental a existência de ferramentas de desenvolvimento que permitam abstrair os detalhes da implementação ao longo das várias fases de *design* de um sistema [1]. Ao processo que engloba todas estas fases, desde a conceção até à obtenção do *hardware* final, dá-se o nome de *design flow* [4], [8], [9].

O *design flow* incorpora várias e diferentes tipos de ferramentas EDAs (*Electronic Design Automations*) [10]. Para além destas ferramentas, as fases envolvidas no *design flow* também variam dependendo da plataforma alvo. Uma comparação, sob a forma de ilustração, entre o *design flow* de um ASIC e de uma FPGA é disponibilizada na Figura 2.

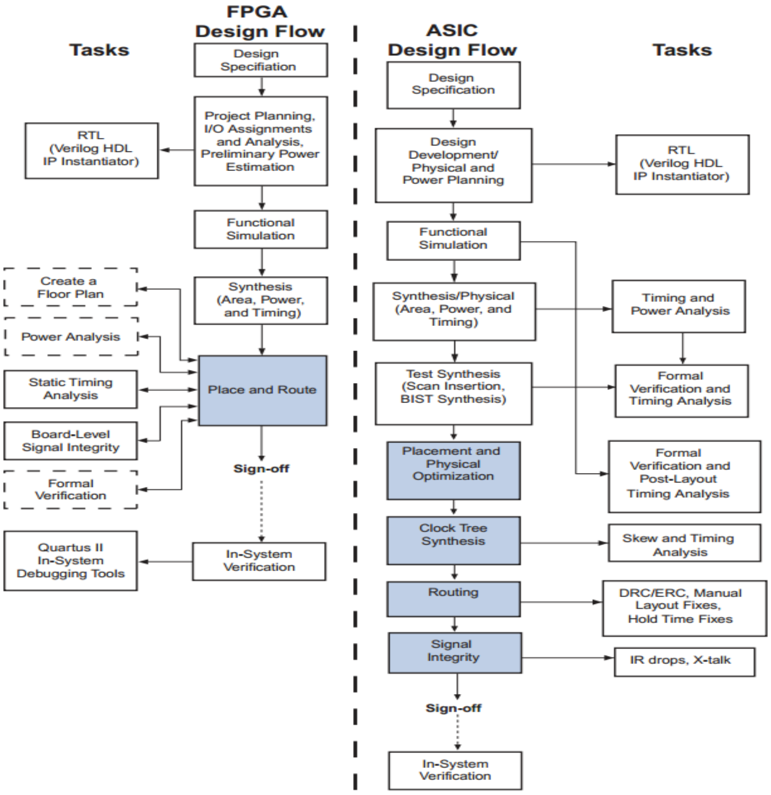


Figura 2 - FPGA *design flow* VS ASIC *design flow* [11]

De seguida são apresentados, com maior detalhe, cada um destes processos, salientando as suas semelhanças e principais diferenças.

2.2.1. ASIC *Design Flow*

De acordo com Kummuru et. al [12], o primeiro passo no processo de desenvolvimento de um ASIC é a especificação do sistema. Esta fase permite definir a funcionalidade, os objetivos e restrições do *design*, as tecnologias de fabrico e as técnicas de *design* a utilizar. Um estudo detalhado das especificações do sistema é bastante importante, porque como referido por Kaur [8], permite definir as interfaces do *design*.

Após concluído o processo de especificação dá-se início ao processo de *design* que consiste em descrever o sistema de acordo com as especificações anteriormente definidas. É nesta fase que se define a arquitetura do ASIC [8], [12]. Segundo Kummuru et. al [12], esta descrição é normalmente realizada utilizando diagramas de blocos ou HDL (*Hardware Description Language*), sendo o VHDL e/ou Verilog as linguagens mais utilizadas [12], [13]. A utilização de um diagrama de blocos permite desenvolver um *design* a um nível de abstração superior oferecendo maior flexibilidade se for necessário efetuar alterações no sistema. No entanto, apesar da utilização de HDL exigir maior esforço e fornecer menor flexibilidade, este tipo de linguagens permitem o desenvolvimento de sistemas mais otimizados. É através da HDL que se obtém o RTL (*Register Transfer Level*) que descreve o sistema. O processo de *design* é concluído com uma fase de simulações comportamentais para verificar se o sistema se comporta como esperado [8], [12]. O processo de simulação recebe como *input* o RTL lógico desenvolvido e disponibiliza como *output* um conjunto de ondas que representam os sinais utilizados pelo sistema [8].

Concluído o processo de simulação inicia-se o processo de síntese do sistema que cria uma descrição do sistema ao nível da porta lógica e um ficheiro *netlist* [12]. A primeira fase deste processo consiste na passagem do RTL lógico para lógica combinacional [8]. Basicamente permite a construção do esquemático de *hardware* do sistema. Segue-se uma fase de otimização do sistema em termos de atrasos, caminho crítico, e área utilizada. O próximo passo é a construção do netlist, contendo uma descrição completamente estrutural, construída apenas por *standard cells*, um bloco básico usado na construção dos ICs (*Integrated Circuits*). Algumas das ferramentas utilizadas para esta etapa são o *Design Compiler*, da Synopsys, ou o *RTL Compiler*, da Cadence. Estas recebem uma descrição RTL do *hardware* e uma biblioteca de *standard cells* e produzem um *netlist*. Este processo é depois seguido de uma simulação pós-síntese para analisar se todas as restrições de tempo são respeitadas [12].

Depois de obter e testar o *netlist* passa-se à fase de implementação do mesmo. Esta fase pode dividir-se em três etapas (i) *Floor Planning*, (ii) *Placement* e (iii) *Routing*, tendo como *output* um ficheiro do tipo GDSII, que posteriormente será utilizado no fabrico do ASIC [12].

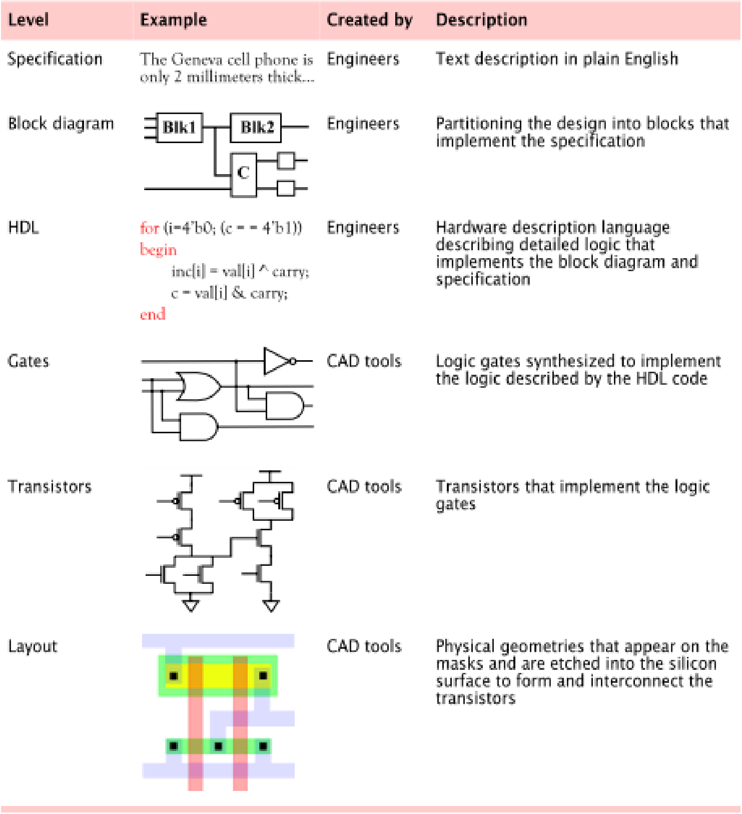
O processo de *Floor Planning* permite definir a localização de cada bloco lógico que constitui o sistema e escolher a melhor forma de os agrupar. O principal objetivo é obter a menor área possível, o menor tempo de atraso e o melhor desempenho. Estes são requisitos contraditórios e difíceis de otimizar simultaneamente. Dependendo da aplicação, alguns destes terão maior ou menor relevância sendo a sua gestão realizada de forma iterativa [14].

O *Placement* consiste no processo de definir a localização física de cada uma das *standard cells* na *die* de silício. Este passo é bastante importante por influenciar métricas como desempenho, rotas possíveis, dissipação de calor, tamanho das ligações e consumos energéticos do *design*. O *Placement* é uma etapa crítica no processo de desenvolvimento de uma ASIC e com o aumento da complexidade dos sistemas torna-se cada vez mais complicado garantir um *Placement* de alta qualidade. De acordo com Cheng [15], uma forma de ultrapassar esta complexidade é realizar o *Placement* em três passos. Primeiro realiza-se um *Placement* global que tem por objetivo distribuir as células de uma forma geral, mesmo que violem algumas restrições enquanto se mantem uma visão global da *netlist*. Segundo, eliminam-se todas as violações das restrições tornando o *Placement* realizado legal, movendo as células localmente. Por fim, através do rearranjo das células, ou grupos de células, tenta otimizar-se o *Placement* final.

Após a conclusão do processo de *Placement* segue-se a definição da *Clock Tree* do sistema. Esta tarefa é uma vez mais, uma tarefa bastante complexa com o objetivo de garantir que o sinal de relógio chega a todos os flip-flops ao mesmo tempo.

Por fim, o processo de *Routing* visa estabelecer as ligações entre todos os blocos e as *nets*, normalmente denominado de *global Routing* e finalmente, estabelecer as ligações entre todas as *nets*, na literatura denominado de *Detailed Routing*. A implementação termina com um conjunto de verificações para determinar se o *layout* foi construído de acordo com as regras especificadas [16].

A Figura 3 descreve sumariamente os processos apresentados e os responsáveis pela execução de cada uma das etapas.



Level	Example	Created by	Description
Specification	The Geneva cell phone is only 2 millimeters thick...	Engineers	Text description in plain English
Block diagram		Engineers	Partitioning the design into blocks that implement the specification

HDL	for (i=4'b0; (c = = 4'b1))
	begin
	inc[i] = val[i] ^ carry;
	c = val[i] & carry;
	end
		Engineers	Hardware description language describing detailed logic that implements the block diagram and specification

Gates		CAD tools	Logic gates synthesized to implement the logic described by the HDL code
Transistors		CAD tools	Transistors that implement the logic gates
Layout		CAD tools	Physical geometries that appear on the masks and are etched into the silicon surface to form and interconnect the transistors

Figura 3 – Overview do processo de *Design Flow* de um ASIC

2.2.2. FPGA *Design Flow*

De acordo com o autor em [13], as ferramentas de desenvolvimento utilizadas no *design flow* de um sistema, fornecidas pelos diversos fabricantes de FPGA, só dão suporte após existir um *design* pronto a ser traduzido para *hardware*, recorrendo a uma HDL. Antes da utilização destas é necessário definir todas as especificações do sistema, e posteriormente, definir o *design* do sistema, à semelhança do que acontece no *design flow* de ASICs.

Apesar de, no conjunto de empresas que desenvolvem ferramentas de EDA, cada uma se basear nas suas ferramentas de desenvolvimento, todas elas são muito semelhantes em termos de funcionalidades, pois em termos genéricos, todas elas se baseiam no mesmo *design flow*.

Na Figura 4 é possível visualizar o *design flow* de FPGA recorrendo ao ambiente de desenvolvimento ISE da Xilinx.

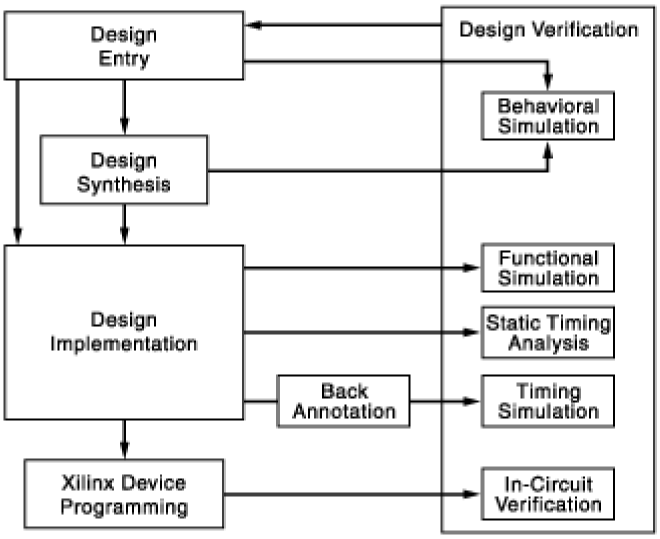


Figura 4 - *Design flow* do ISE da Xilinx [17]

De acordo com [13], as etapas mais comuns envolvidas no *design flow* de sistemas baseados na tecnologia de FPGA são (i) *design*, (ii) síntese, (iii) *place* & *route* e, por fim, (iv) geração do *bitstream*.

A fase de *design* consiste na criação de uma representação lógica do sistema, normalmente recorrendo a HDL. Depois de definido o *design*, segue-se a fase de síntese [17]. A partir daqui podem ser retiradas vantagens do suporte disponibilizado pelas ferramentas individuais de cada fabricante de FPGA. Depois de selecionada a família e o FPGA em específico, as ferramentas de síntese criam uma *netlist* que se baseia nas primitivas do fabricante de forma a implementar o comportamento especificado nos ficheiros HDL fornecidos. Durante a criação da *netlist*, a maioria das ferramentas executam algumas operações adicionais, como por exemplo a otimização lógica, com o objetivo de melhorar o desempenho temporal do sistema e obter uma *netlist* mais eficiente [13].

A fase que se segue, *Place & Route*, começa por definir a localização no *chip*, para cada primitiva presente na *netlist*, e segue estabelecendo ligações entre todas as primitivas colocadas, garantindo as restrições temporais previamente impostas [13]. Grande parte do esforço nesta fase é realizado através das ferramentas de desenvolvimento disponibilizadas pelo fabricante, tendo o utilizador que definir apenas o mapeamento entre os pinos da FPGA e os sinais de entrada e saída do módulo principal, ao contrário do que sucede no ASIC, como já mencionado.

Terminado o processo anterior segue-se a geração de um ficheiro *bitstream*, que contem as definições para cada um dos elementos programáveis presentes na FPGA. Este ficheiro é depois enviado para a FPGA alvo, programando-a.

A primeira fase do *design flow* baseado em tecnologia de FPGA, exige uma força laboral elevada. Todas as restantes fases são fortemente apoiadas pelas ferramentas de desenvolvimento disponibilizadas pelos fabricantes de FPGA.

Pode então concluir-se que o *design flow* de ASICs e FPGA são bastante semelhantes, como é possível verificar na Figura 5. São executadas as mesmas etapas em ambos os casos, no entanto, no caso do ASIC há um esforço de engenharia superior devido a algumas etapas não serem automáticas como se verifica nas FPGA. Por este motivo as FPGA são cada vez mais utilizadas em detrimento dos ASIC, uma vez que diminuem os riscos associados ao desenvolvimento, oferecem cada vez mais recursos e favorecem o cumprimento de metas como a do *time-to-market* [11], [9].

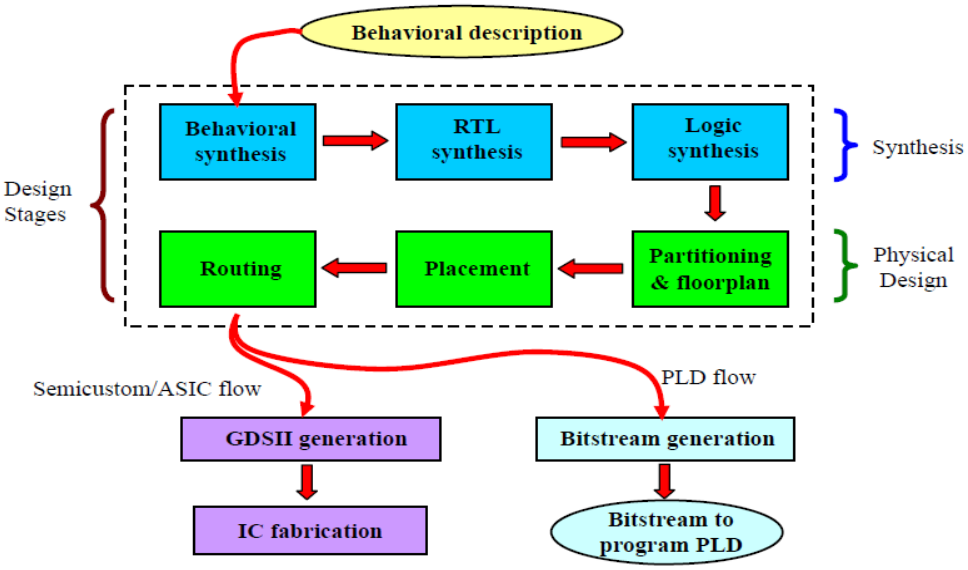


Figura 5 - *Design flow* Geral [9]

O facto de o processo de *design* ser a única etapa, no desenvolvimento de um sistema recorrendo à tecnologia de FPGA, que exige um esforço maior, é apontado por Serrano [13] como a razão para o aparecimento, nos últimos anos, de ferramentas que permitem um maior nível de abstração na fase de *design*. Para isso, estas ferramentas oferecem a possibilidade de definir o sistema recorrendo a blocos, que depois serão analisados por um *software* que é responsável pela sua conversão recorrendo a linguagens de HDL. Comparativamente com a tecnologia FPGA, a utilização de tecnologia ASIC exige um conhecimento técnico e uma força laboral elevada, uma vez que, apesar de existirem ferramentas de apoio, a obtenção de um sistema de elevado desempenho só é possível através de um estudo exaustivo e pormenorizado de todos os aspetos do ASIC.

Nas secções seguintes serão apresentadas algumas ferramentas utilizadas para o desenvolvimento do *design* dos sistemas recorrendo a blocos gráficos. Para além de um grau de abstração superior, as ferramentas descritas são compatíveis com o *standard* IEEE 1685 IP-XACT, o que confere aos IPs gerados uma maior portabilidade entre ambientes de desenvolvimento e consequente independência relativamente ao seu produtor.

2.2.3. Kactus2

O Kactus2 é um Ambiente de Integrado de Desenvolvimento que visa facilitar a comunicação entre programadores de *hardware* e de *software* [18]. Centra-se essencialmente no *design flow* do projeto, com o objetivo de possibilitar a reutilização de componentes. Esta ferramenta segue o paradigma *general plataform based design*, associado ao uso de meta-dados. A utilização de meta-dados para representar todos os componentes e sistemas criados garante a interoperabilidade entre plataformas e ferramentas. Esta independência entre plataformas é conseguida porque o Kactus2 tem como base o IEEE 1685 IP-XACT *metadata*.

Numa tentativa de colmatar as limitações apresentadas pelo IEEE 1685/IP-XACT, esta ferramenta apresenta algumas extensões ao *standard*, que permitem contemplar o desenvolvimento de IPs de *software*. Deste modo, fazendo uso campo *vendorExtensions* presente no *standard* [18], passa a ser possível representar todo o tipo de *software* através de meta-dados, permitindo com isto, agilizar o processo de definição da comunicação entre o *hardware* e o *software*. A Figura 6 ilustra as extensões adicionadas pela equipa de desenvolvimento do Kactus2 comparativamente com a versão original do IP-XACT. A azul é possível visualizar-se as componentes abrangidas pelo *standard* IP-XACT. O cubo representado a verde assinala as componentes que as extensões do Kactus2 introduziram.

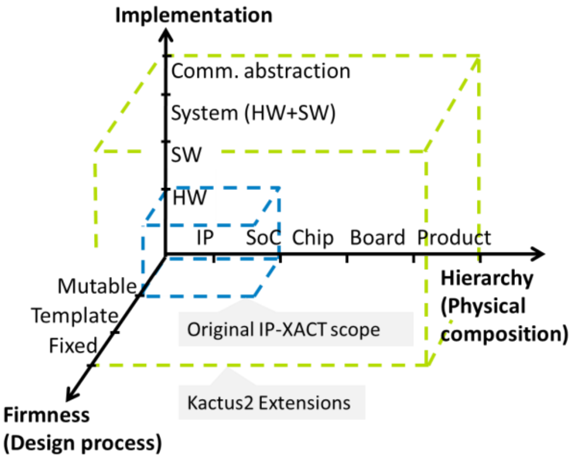
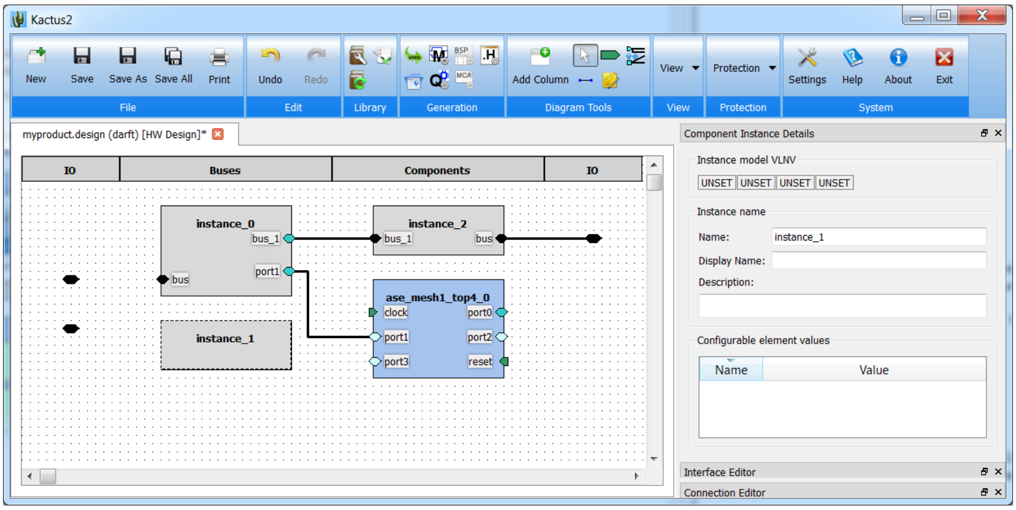


Figura 6 - Extensões do Kactus2 [18]

O processo de desenvolvimento e criação de um novo componente ou sistema, utilizando esta plataforma, segue um fluxo de desenvolvimento também ele baseado nos meta-dados [18]: primeiramente os componentes são encapsulados e separados do código fonte que os representa (em VHDL, C, etc.), sendo esta representação inserida através de *links* presentes no ficheiro com os meta-dados (ficheiro IP-XACT); de seguida os ficheiros com os meta-dados são utilizados para a criação de sistemas mais complexos, constituídos por um ou mais componentes; por último, o *design* final produzido pode ser interpretado como um conjunto de instruções para a geração dos executáveis. Na Figura 7 é apresentado o visual oferecido para o desenvolvimento através da ferramenta Kactus2.



UNSET UNSET UNSET UNSET
Name: instance_1

Figura 7 – Desenvolvimento de um sistema no Kactus2 [18]

Em suma, a ferramenta do Kactus2 permite-nos [18]: (i) criar componentes na forma de meta-dados de acordo com o *standard* IP-XACT; (ii) importar componentes e bibliotecas que respeitem o *standard* IP-XACT; (iii) importar/exportar os meta-dados referentes a todos os componentes presentes na biblioteca do Kactus2; (iv) manter a consistência dos dados; (v) elaborar documentação sobre os componentes criados; (vi) gerar *templates* para o código dos componentes com base nos seus meta-dados; (vii) desenhar canais de comunicação entre IPs; (viii) mapear *hardware* e *software*; (ix) configurar os parâmetros de cada componente; (x) desenhar sistemas e geração dos meta-dados associados.

O conceito adotado pelo Kactus2 é um pouco diferente quando comparado com outras ferramentas existentes no mercado. Esta diferença é visível através da comparação dos seus *outputs*, enquanto o *output* do Kactus2 é constituído por meta-dados, outras ferramentas semelhantes têm como *output* ficheiros executáveis. Na verdade o Kactus2, para além de não permitir a geração de executáveis e *bitstreams*, também não permite a definição da funcionalidade de um IP [18]. Para isso é necessário recorrer a outras ferramentas para editar os ficheiros de código.

Em resumo, o Kactus2 centra-se essencialmente na gestão organizada de uma biblioteca de meta-dados sobre o formato do IEEE1685/IP-XACT, favorecendo o processo de *design*, a fase inicial de criação do sistema alvo, e a documentação de componentes IP/sistemas.

2.2.4. MAGILLEM 4.0

A MAGILLEM [19] afirma que a sua ferramenta proporciona aos utilizadores uma plataforma que reduz drasticamente o tempo de desenvolvimento de sistemas. A juntar a esta vantagem, proporciona também independência entre fornecedores de ferramentas EDAs, uma vez que também se baseia na utilização do *standard* do IEEE 1685 IP-XACT.

Convém salientar que, a equipa da MAGILLEM contribuiu fortemente para a definição do *standard* IP-XACT, e de acordo com informação presente no seu *site* [19], é a única ferramenta que suporta de forma completa todas as versões deste, incluindo a última versão definida no SPIRIT Consortium.

MAGILLEM 4.0 integra ainda um conjunto de ferramentas que permite a gestão de uma base de dados compatível com o *standard* IP-XACT, e um ambiente completo de *design*, que inclui funcionalidades de *debug* e a possibilidade de executar geradores externos ao software da MAGILLEM [19].

A MAGILLEM defende que a sua ferramenta não é apenas mais uma ferramenta, mas sim um ambiente que possibilita a interoperabilidade entre ferramentas, de forma contínua e automática, como pode ser visto através da Figura 8. Na sua constituição, esta ferramenta tem como base a máquina virtual Java e é distribuído na forma de um *plug-in* do Eclipse, sendo assim compatível com todo o tipo de arquiteturas e sistemas operativos do *Host* [19].

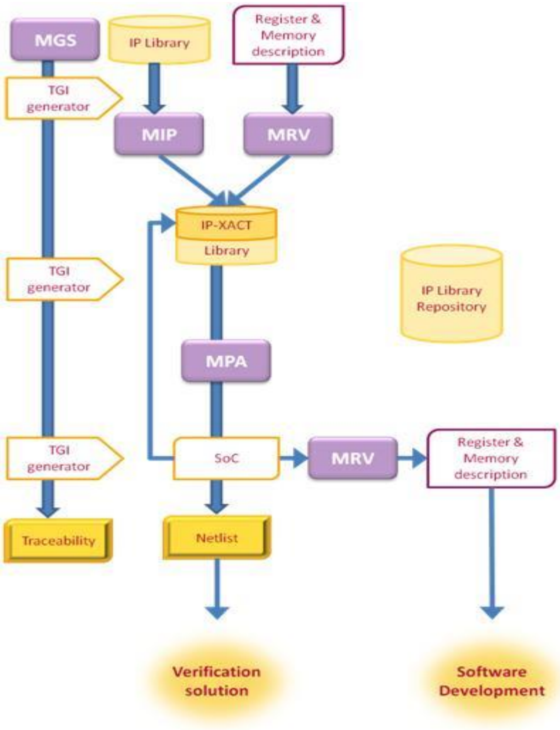


Figura 8 - *Design flow* do Magillem 4.0 [20]

O *plug-in* presente no Eclipse reúne todo um conjunto de ferramentas constituído por: (i) um gestor de IP-XACT; (ii) uma plataforma de assemblagem; (iii) um completo ambiente de desenvolvimento; (iv) e uma ferramenta para controlo de fluxo do processo de *design* [21]. Esta arquitetura é visível na Figura 9, e encontra-se dividida em três partes, sendo elas: (i) o módulo para importar IPs; (ii) o módulo para construir o *design*; e (iii) os recursos para realizar o controlo de fluxo do processo de *design*.

O primeiro módulo garante a geração de ficheiros XML, respeitando o *standard* do IP-XACT e assegurando a deteção de erros. O segundo módulo oferece a possibilidade de construir um sistema completo, através de um esquemático, ou fazendo uso de um *batch* de comandos. Por último as ferramentas para o controlo de fluxo permitem a criação de geradores compatíveis com o IP-XACT, para extração de informações relevantes da base de dados ao longo de um determinado processo do fluxo de desenvolvimento [19].

Resumidamente, esta ferramenta, o MAGILLEM 4.0, é uma ferramenta inovadora que permite importar, realizar o *design* e controlar o fluxo de desenvolvimento de IPs.

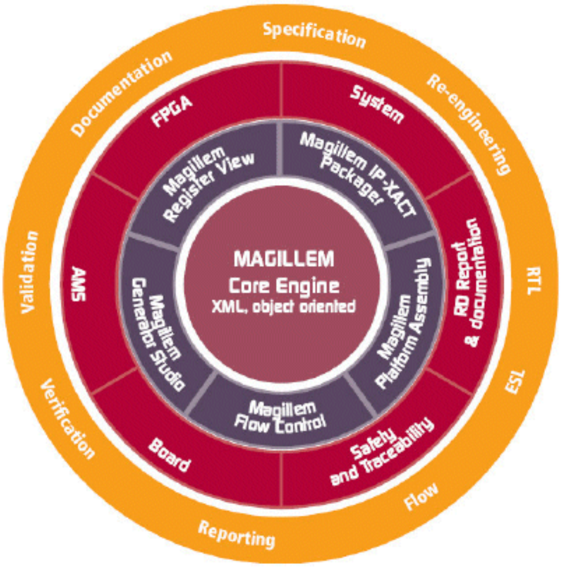


Figura 9 - Arquitetura do MAGILLEM 4.0 [21]

2.2.5. Synopsys coreTools

A Synopsys é uma empresa que disponibiliza um conjunto de produtos e serviços inovadores que permitem acelerar todo o processo de design no mercado da eletrónica, sendo um dos líderes no desenvolvimento de ferramentas de EDA e no desenvolvimento de IPs, contando com um conjunto consistente de ferramentas para importação, *design* e verificação dos mesmos. De acordo com o *datasheet* da Synopsys [22], este conjunto de ferramentas permite aos utilizadores maximizar a sua produtividade, reduzindo até 60% no tempo de desenvolvimento. O coreTools é constituído por três ferramentas: (i) coreBuilder; (ii) coreAssembler; (iii) coreConsultant [22].

O primeiro, o coreBuilder, consiste num conjunto robusto de ferramentas que permite definir IPs através de ambientes gráficos ou através de ambientes baseados em script de comandos.

Permite ainda exportar todo o tipo de vistas, contendo diferentes aspetos, necessários às diferentes equipas de desenvolvimento envolvidas no processo de *design*. Tudo isto contribui para uma redução do custo de suporte ao IP e para uma melhoria na qualidade do mesmo, assegurando em simultâneo a completa compatibilidade com o *standard* IP-XACT [2].

O coreAssembler permite a geração automática do RTL totalmente configurado, assim como toda a documentação relativa ao bloco implementado, incluindo os detalhes das configurações do sistema e um conjunto de *testbenchs* realizáveis no sistema. Quando combinado com o coreBuilder as duas ferramentas permitem conjugar diversos subsistemas naquilo a que a Synopsys chama de coreKit. Estes permitem a criação de sistemas configuráveis prontos para um determinado mercado alvo. Por fim o coreAssembler é responsável por gerar os ficheiros IP-XACT relativos ao *design* do sistema. Na Figura 10 é possível visualizar o processo de criação de um coreKit.

O coreConsultant é um pacote de utilitários para configuração, validação e implementação de blocos individuais de IPs incluído no coreBuilder. Tal como as restantes ferramentas, o coreConsultant é responsável por criar os ficheiros IP-XACT dos blocos de IPs [22].

Todos os ficheiros gerados em torno dos IPs desenvolvidos por estas ferramentas podem depois ser utilizados por qualquer plataforma compatível com o IP-XACT, tornando-os assim independentes da linguagem e do ambiente de desenvolvimento.

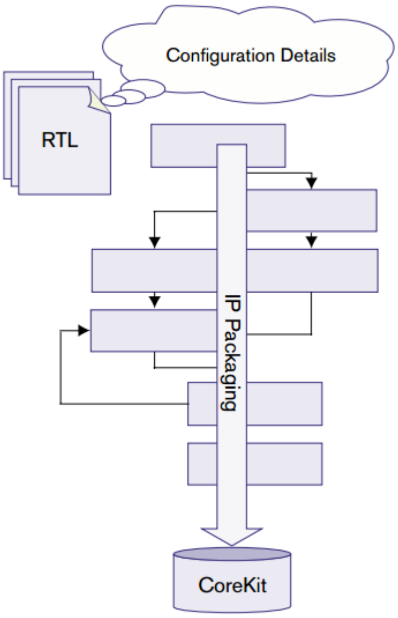


Figura 10 – Criação de um coreKit [22]

2.3. Ambientes Integrados de Desenvolvimento de *Software*

Um Ambiente Integrado de Desenvolvimento (ou IDE – *Integrated Development Envoronment*, na nomenclatura inglesa) é uma ferramenta de *software* criada para facilitar o processo de desenvolvimento de sistemas de *software* e/ou *hardware* [23]. Disponibiliza um conjunto de ferramentas que, normalmente, integram: (i) um editor de texto que permite ao programador inserir o código respetivo da aplicação alvo; (ii) um compilador/sintetizador, que é responsável pela geração de código executável e *bitstream* para a máquina alvo; e (iii) um simulador com funcionalidade de depuração, que permite facilitar o processo de verificação e correção de erros [24], [25].

No entanto, hoje em dia, os ambientes de desenvolvimento oferecem cada vez mais funcionalidades aos utilizadores com o objetivo de agilizar as mais variadas tarefas desde o controlo de versões e do código desenvolvido à gestão da própria equipa de desenvolvimento.

Atualmente a lista de plataformas de desenvolvimento que estes IDEs integram é quase interminável e grande parte deles dá suporte ao desenvolvimento sobre diversas linguagens de programação e para diversas plataformas.

De acordo com Jacobs [26], os ambientes integrados de desenvolvimento mais utilizados atualmente são:

- Visual Studio, que oferece suporte para a programação em diversas linguagens, entre elas o C, C++, C#, J#, javascript, XAML, HTML5;

- Eclipse, que oferece nativamente o suporte para a programação em Java, e que no entanto, através da instalação de *plug-ins* é possível utilizá-lo para programar em C, C++, Ruby, etc;

- Netbeans, que oferece suporte para a programação em java, PHP e C++.

No entanto, o Visual Studio da Microsoft continua a afirmar-se como uma das ferramentas comerciais mais utilizadas, principalmente para o desenvolvimento de aplicações que têm como plataforma alvo os ambientes de sistemas operativos baseados em Windows [27].

2.3.1. Importância de um Ambiente Integrado de Desenvolvimento

A importância dos Ambientes Integrados de Desenvolvimento cresceu ao longo dos anos devido principalmente ao aumento da complexidade do código desenvolvido [27]. A sua utilização oferece diversas vantagens nomeadamente, a redução do tempo de desenvolvimento e deteção de erros de codificação quando comparado com ferramentas CLI (*Command Line Interface*).

Antes do seu aparecimento os programadores tinham a necessidade de conhecer uma grande quantidade de bibliotecas de funções na linguagem pretendida, ou então, tinham de conhecer um grande número de classes e/ou funções membro nos casos das linguagens C++/Java [28]. Os processos de depuração eram bastante morosos, tornando-se cada vez mais complexos com o aumento do tamanho do programa, cuja apresentação não permitia uma leitura tão fluída como a que se consegue atualmente, recorrendo aos mecanismos de apresentação estruturada, representação gráfica e coloração de código [28]. A utilização de um bom Ambientes Integrados de Desenvolvimento resulta normalmente numa maior produtividade, geração e reutilização de código, e consequentemente, maior lucro para a empresa. Este não só permite reduzir o tempo de desenvolvimento, como numa fase inicial, também simplifica o processo de aprendizagem de uma nova linguagem. Estas melhorias são possíveis devido a algumas funcionalidades oferecidas pelas plataformas de desenvolvimento atuais. Stratulat [28] apresenta algumas destas vantagens:

- Coloração de acordo com a sintaxe da linguagem, facilitando o processo de leitura e análise do código produzido;

- Funcionalidade de *autocomplete*, evitando que o programador tenha de conhecer toda a sintaxe e/ou conjunto de bibliotecas inseridas num determinado projeto. Normalmente são apresentadas várias propostas de completação, com os respetivos parâmetros de entrada e o tipo de retorno, e o programador escolhe qual das opções pretende utilizar;

- Ferramenta de pesquisa, com auxílio na identificação de segmentos de código ao longo de um projeto global. Por exemplo, torna-se muito mais fácil identificar todos os locais onde é utilizada uma determinada variável;

- Funcionalidades de *Refactoring*, que permitem a reestruturação do código sem alterar o seu comportamento exterior, melhorando atributos não funcionais, reduzindo a complexidade e facilitando manutenção do código criado;

- *Code snippets*, que evitam as constantes repetições de código;

- Controlo de versões, que promove uma maior organização do projeto e permite controlar a sua evolução, fornecendo funcionalidades como a criação de *patches* e *diffs* entre duas ou mais versões. Estas funcionalidades permitem reverter o estado de um projeto, com o objetivo de deteção e verificação de erros, uma vez que este tipo de funcionalidade fornece informações sobre as últimas alterações efetuadas no código, a data em que foram realizadas e quem as efetuou;

- Centralização, talvez uma das vantagens mais importantes de um IDE. Oferece todas as ferramentas necessárias numa única plataforma.

Apesar de todas as vantagens oferecidas pelo uso de IDEs, a verdade é que existe alguma dificuldade em encontrar uma que ofereça ao programador um conjunto completo, com tudo aquilo que ele precisa.

No subcapítulo seguinte descreve-se a evolução dos Ambientes Integrados de Desenvolvimento ao longo do tempo e as mudanças que essas evoluções implicaram.

2.3.2. Evolução dos Ambientes Integrados de Desenvolvimento

A utilização de Ambientes Integrados de Desenvolvimento tornou-se possível a partir do momento em que se começou a desenvolver com recurso a uma consola ou terminal. Os sistemas anteriores não ofereciam essa possibilidade uma vez que os programas eram desenvolvidos recorrendo a fluxogramas e a cartões perfurados [29].

O primeiro IDE para *software* foi criado pela Softlab Munich em 1975 e dava pelo nome de Maestro I [30]. Inicialmente era designado por PET (*Program development System Terminal*) mas o nome veio a ser alterado por entrar em conflito com o sistema computacional da

Commodore, o PET. A primeira instalação do Maestro I nos EUA ocorreu na Boing em 1979 e rapidamente conseguiu chegar às 22000 instalações em todo o mundo.

Em 1989 a Hewlett-Packard, vulgarmente designada por HP, lançou o SoftBench, o primeiro IDE a operar sobre um sistema operativo UNIX e que já utilizava o conceito de *plug-in* [31]. Inicialmente suportava compilação de linguagem C, e posteriormente, as sucessivas versões lançadas pela empresa foram introduzindo outras linguagens como o C++ e o Cobol. Na Figura 11 é possível visualizar a evolução das várias versões do SoftBench ao longo dos anos. Os principais objetivos deste projeto eram: (i) oferecer ao utilizador um ambiente orientado a tarefas, para que o utilizador se pudesse concentrar no trabalho que tinha de realizar sem se preocupar com a forma de o concretizar; (ii) suportar a interoperabilidade entre ferramentas; (iii) suportar um ambiente de desenvolvimento distribuído; (iv) dar suporte à integração de outras ferramentas de controlo do ciclo de vida do *software*; (v) suportar o desenvolvimento de *software* em equipas; (vi) potencializar as ferramentas existentes; (vii) abranger as diferentes fases do desenvolvimento de um projeto; (viii) construir respeitando os *standards* existentes.

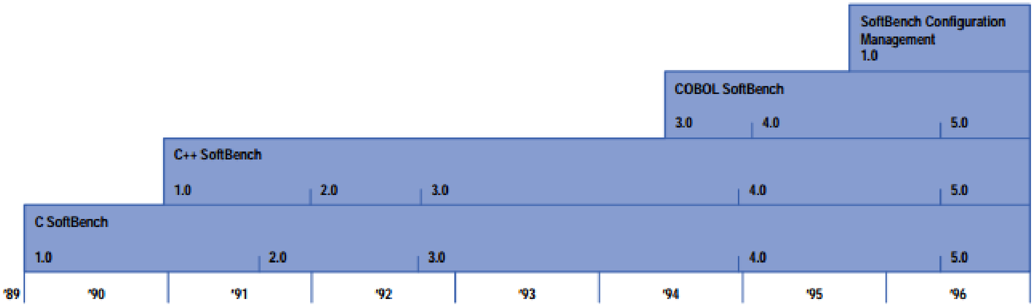


Figura 11 - Evolução do ambiente de desenvolvimento da SoftBench [31]

Estas características serviram mais tarde de inspiração para as novas ferramentas desenvolvidas e utilizadas atualmente como por exemplo o *software* Eclipse [32].

Comparativamente com as ferramentas atuais, estas sofreram uma grande evolução oferecendo cada vez mais funcionalidades, facilitando não só o processo de desenvolvimento de sistemas de *software* mas também de *hardware* até ferramentas de a gestão/distribuição de trabalho pela equipa de desenvolvimento, incluindo ferramentas de controlo de versões, mecanismos de compilação automática e periódica, e controlo de deadlines em alinhamento com a distribuição de esforços pelos elementos da equipa [33]. O Anexo A deste documento apresenta uma explicação detalhada de alguns dos Ambientes Integrados de Desenvolvimento mais utilizados na atualidade.

2.4. Conclusões

Após analisados todos os fatores referidos ao longo deste capítulo, optou-se pela utilização do Visual Studio como Ambiente Integrado de Desenvolvimento da *framework* proposta por esta dissertação. O facto de o autor já se encontrar familiarizado com o IDE e respetivas plataformas de desenvolvimento, aliado à tendência atual do contexto empresarial, visando o Visual Studio como uma das ferramentas comerciais mais utilizadas, constituem os principais motivos pela sua escolha. No entanto, as avançadas funcionalidades de depuração e meta informação que este integra, incluído nesta a completação de código, foram igualmente importantes, uma vez que se mostraram bastante úteis no desenvolvimento de projetos de grandes dimensões, em outros trabalhos complementares a esta dissertação.

Escolhido o ambiente de desenvolvimento optou-se pela utilização da linguagem de programação C# para a implementação da ferramenta proposta. Uma vez mais, a familiarização prévia do autor com esta linguagem de programação, aliada à sua potencialidade e flexível integração de múltiplas plataformas de desenvolvimento, como o .Net, foram fatores importantes no processo de escolha.

Todas as decisões supracitadas foram tomadas com a complexidade do projeto em mente e com o objetivo de reduzir a relação tempo/esforço. A utilização do Visual Studio permitiu reduzir drasticamente a curva de aprendizagem do autor, e a escolha de C# como linguagem de programação torna possível o uso de um vasto leque de bibliotecas gráficas e documentação *online*, facilitando o processo de implementação. Deste modo, reduz-se o período de aprendizagem, alargando o tempo disponível para o processo de implementação efetiva da ferramenta proposta.

Capítulo 3

ESPECIFICAÇÃO E *DESIGN* DO SISTEMA

Este capítulo aborda todas as etapas de análise e *design* realizadas no desenvolvimento da *framework* para repositório IP-XACT. Os resultados deste estudo fundamentam a implementação da referida *framework*. O sistema foi concebido seguindo uma abordagem modular e a organização deste será explicada, bem como as decisões tomadas que serão também fundamentadas. O capítulo termina com uma descrição das ferramentas usadas na implementação da *framework* e ferramentas externas com as quais interage.

3.1. Funcionalidades e Restrições

Como mencionado no capítulo introdutório o projeto proposto por esta dissertação passa por desenvolver uma *framework* compatível com o *standard* IP-XACT. Importa portanto analisar que funcionalidades e restrições estão implícitas no projeto em causa.

A primeira funcionalidade a suportar é a gestão do *standard*, ou seja, a *framework* tem de ser capaz de ler e escrever documentos no formato IP-XACT. Deve também ser capaz de analisar todos os ficheiros e verificar se existem irregularidades. Ainda no sentido de evitar erros nos ficheiros produzidos, a *framework* não deve permitir a criação de um ficheiro que não cumpra todas as especificações impostas pelo *standard*.

Para organizar os IPs existentes, deve ser criado um repositório com todos os ficheiros relativos ao *standard*. Este repositório deve conter apenas ficheiros considerados válidos pelo IP-XACT.

Como referido no capítulo anterior, uma *framework* oferece normalmente um conjunto de funcionalidades ao utilizador, que vão desde a coloração e completação de código, à depuração e localização de erros no momento da compilação. No entanto, com esta *framework* pretende-se aumentar o nível de abstração do programador, a um nível em que o desenvolvimento do sistema seja realizado o máximo possível recorrendo a blocos que representam IPs do repositório. É claro que o processo de escrita de código é inevitável, mas neste caso assume um papel de menor importância devido a abordagem generativa que foi seguida com a extração dos artefactos de código a partir dos conteúdos dos IPs IP-XACT. Ao nível das verificações, interessa evitar que o programador cometa erros em vez de simplesmente o avisar quando tal acontece. Por esse motivo é importante a implementação de mecanismos de verificação em *design time* que impeçam o utilizador de efetuar as operações inválidas.

Com o aumento da complexidade dos sistemas as metodologias de co-*design* são cada vez mais utilizadas e a sua utilização oferece bastantes vantagens porque permitem que o *hardware* e *software* sejam desenvolvidos em simultâneo, diminuindo o tempo de desenvolvimento e possibilitando a identificação de erros de projeto relativamente cedo no processo de desenvolvimento [34].

Por forma a abranger todo o *design flow*, a *framework* tem de oferecer a possibilidade de gerar, compilar e enviar o código para a plataforma alvo, a partir do *design* realizado. A geração de código terá de ser realizada a partir dos artefactos embutidos nos ficheiros IP-XACT de cada IP (por exemplo, ficheiros C/C++, Verilog, VHDL, etc). Dependendo da plataforma alvo as ferramentas para a compilação variam, pelo que a invocação de ferramentas externas à *framework* é necessária. Desta forma, deve disponibilizar intuitivamente um mecanismo para invocação das ferramentas indicadas para cada ficheiro. Por fim, após gerado e compilado o código, é necessário programar a plataforma alvo. Mais uma vez, este serviço será disponibilizado por uma ferramenta externa à *framework*, pelo que os devidos mecanismos para implementar tal função devem ser garantidos.

Através da implementação de todos os mecanismos supracitados, garantimos que o *design flow* é totalmente coberto pelo sistema a desenvolver, desde o processo de *design* até ao processo de programação da plataforma alvo.

Em suma, a *framework* a desenvolver tem de (i) gerir ficheiros IP-XACT, (ii) oferecer uma interface gráfica para guiar o utilizador ao longo do processo de *design flow* e (iii) gerar e compilar código para uma plataforma específica.

Desta forma podemos dividir o sistema em três módulos distintos. O primeiro módulo é caracterizado por implementar todas as funcionalidades referentes à gestão do *standard* IP-XACT. Ao segundo módulo cabe a responsabilidade de fornecer a interface gráfica ao utilizador. Por fim, o terceiro módulo corresponde a um gerador de código e aos mecanismos de invocação das ferramentas externas. Nas secções seguintes serão apresentados cada um destes módulos em maior detalhe.

3.1.1. Gestor de Repositório para IP-XACT

O módulo da gestão do repositório tem de garantir que todos os documentos presentes no repositório devem assumir um formato compatível com o *standard*, ou seja, todos os ficheiros XML presentes no repositório têm de cumprir as especificações do IP-XACT e todos os ficheiros que não estejam em formato XML tem de estar referenciados pelo menos num ficheiro IP-XACT. Por forma a manter a coerência e a integridade dentro do repositório, sempre que o programa é iniciado deve ser dado ao utilizador a possibilidade de realizar um *scanner* a todas as diretorias do repositório para verificar a validade de cada ficheiro. Se o ficheiro a ser analisado não cumprir os requisitos enumerados pelo IP-XACT o utilizador deve ser notificado. Na Figura 12 é apresentado o diagrama de casos de uso para o primeiro módulo.

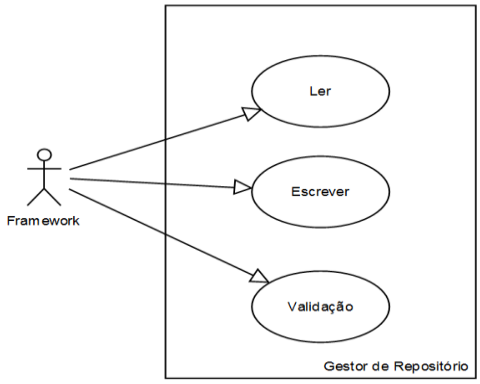


Figura 12 - Diagrama de Casos de Uso do Gestor do Repositório

Por forma a aprofundar a análise de cada um dos casos de uso identificados foram desenvolvidos diagramas de sequência, disponibilizados no Anexo B. A título de exemplo será analisado com maior detalhe o diagrama de sequência correspondente ao processo de arranque da *framework* (Figura 13) no caso em que se invoca a funcionalidade de validação oferecida pelo gestor de repositório.

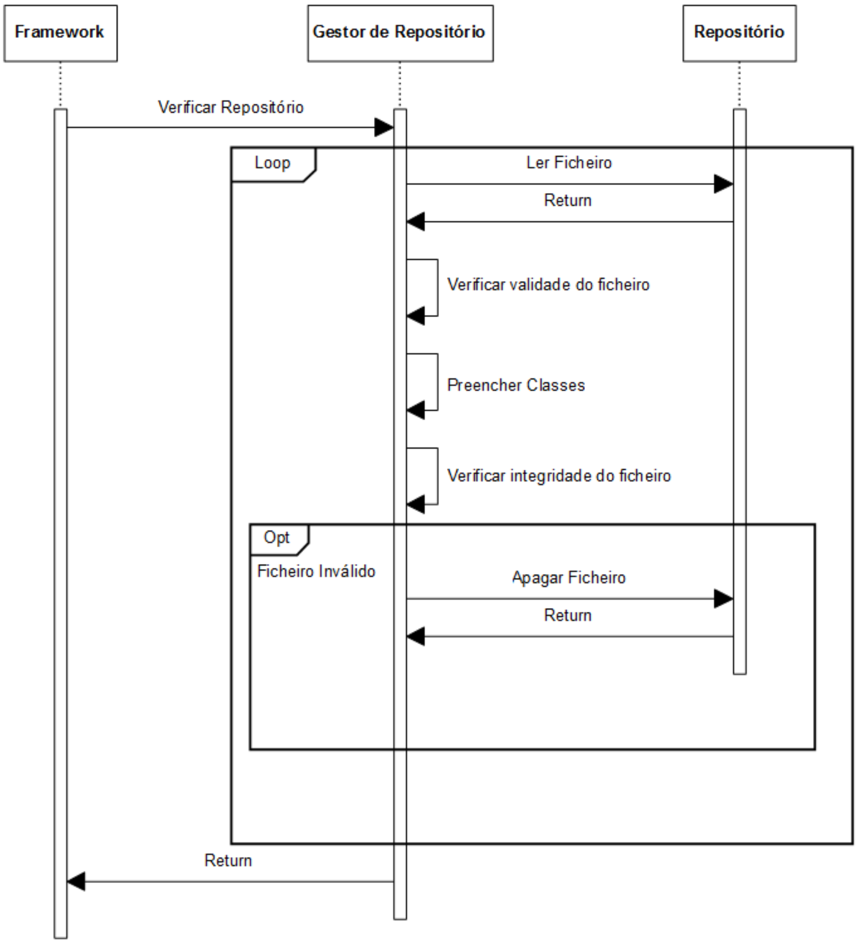
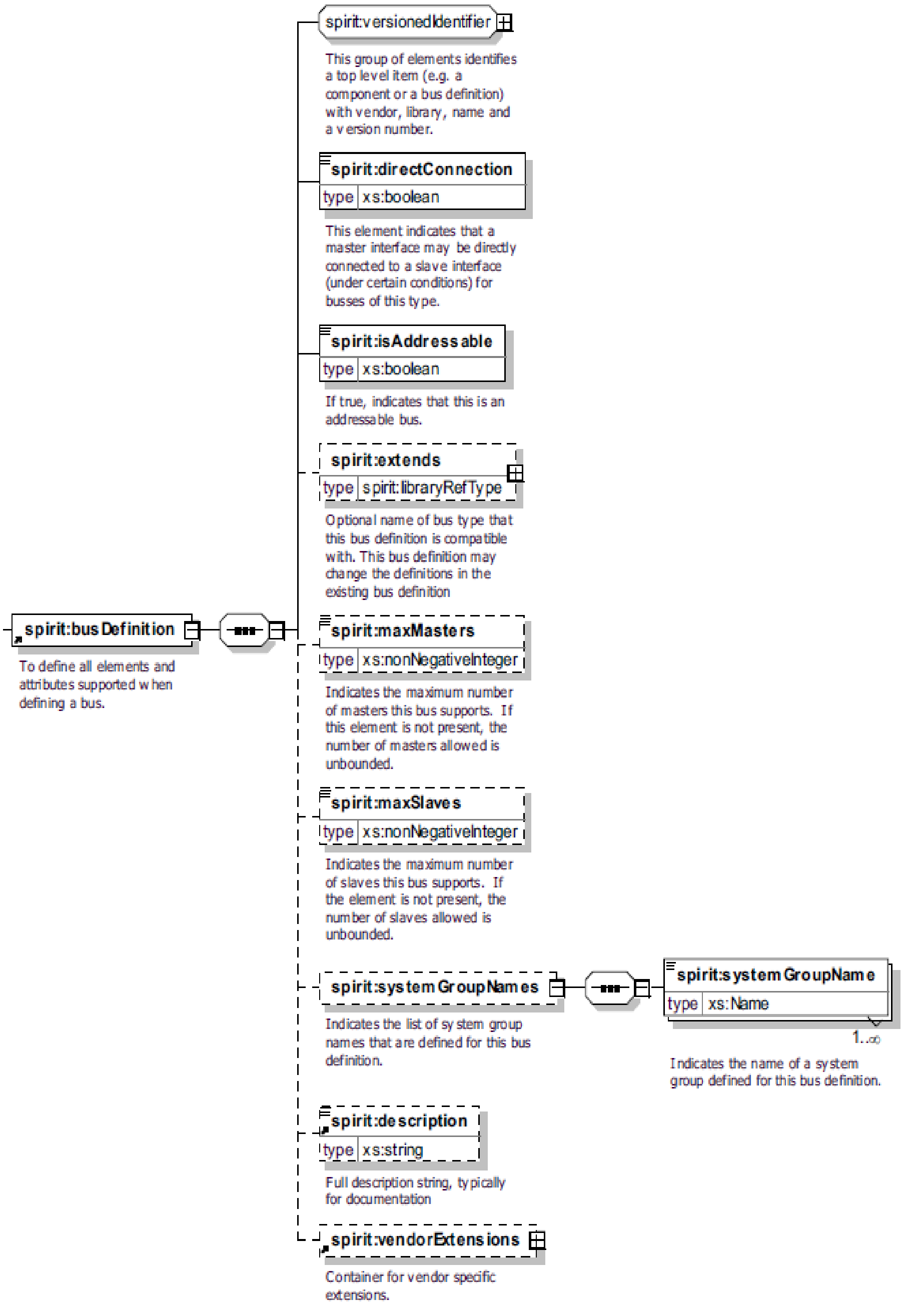


Figura 13 - Diagrama de Sequência: *Start Up*

No momento de arranque da *framework* é enviado ao módulo do gestor de repositório um pedido de verificação a todo o repositório. Ao receber a raiz do repositório o módulo de gestão lê e verifica os ficheiros um a um. O utilizador é notificado caso o processo de verificação indicar que o ficheiro lido não cumpre os requisitos, sendo depois possível apagar esse mesmo ficheiro. Após analisar todos os ficheiros na diretoria e subdiretorias do repositório, o módulo de gestão retorna e a *framework* continua a sua execução normal.

A correta gestão do repositório implica um conhecimento profundo do *standard* em causa. Seguidamente serão descritos apenas os aspetos mais importantes do *standard*, recorrendo a um dos sete elementos de topo do *standard*. A apresentação exaustiva do *standard* na dissertação seria demasiado extensa pelo que se optou por utilizar o *top element busDefinition* para demonstrar todo o processo de análise realizado em torno do *standard*. O processo foi depois repetido para cada um dos restantes seis *top elements* do IP-XACT. A versão do *standard* utilizado para a implementação do gestor do repositório é a versão 1.5.

O *XML Schema* representativo do elemento *busDefinition* é apresentado na Figura 14. Através da análise do *XML Schema* é possível retirar quase toda a informação necessária para implementar as especificações do *standard*. Todos os blocos cujos limites estejam representados a cheio representam blocos obrigatórios, ou seja, o ficheiro IP-XACT que não contenha esses elementos estão incorretos. Os blocos com os limites a tracejado representam elementos opcionais. Outro aspeto a considerar são os blocos de *choice*, que representam uma escolha de 1 de n elementos. No caso de blocos complexos, isto é, blocos compostos por outros blocos, existem duas configurações a analisar. No primeiro caso a inclusão do bloco opcional obriga a que depois todos os blocos obrigatórios que o constituem estejam presentes no documento. No entanto, a inclusão de um bloco opcional continua a não ser obrigatória mesmo que este contenha blocos obrigatórios na sua constituição. No segundo caso, a inclusão do módulo principal é obrigatória assim como a inclusão de todos os blocos obrigatórios que o constituem. Os blocos retangulares representam elementos e por isso, no momento da construção do diagrama de classes darão origem a uma classe ou variável, dependendo se são um elemento de tipo composto ou simples respetivamente. Os blocos octogonais representam grupos de elementos. No momento da construção do diagrama de classes estes blocos podem ser removidos uma vez que não acrescentam qualquer informação ao *standard*. Para além destes grupos, existem grupos de atributos. Tal como acontece com os grupos representados pelo bloco octogonal, os grupos de atributos também não têm representação no diagrama de classes. No entanto, os elementos pertencentes a um grupo de atributos dão origem a variáveis que terão de ser tratadas de forma diferente no momento da escrita e leitura dos ficheiros XML onde aparecerem referenciadas.



This group of elements identifies a top level item (e.g. a component or a bus definition) with vendor, library, name and a version number.

This element indicates that a master interface may be directly connected to a slave interface (under certain conditions) for busses of this type.

If true, indicates that this is an addressable bus.

Optional name of bus type that this bus definition is compatible with. This bus definition may change the definitions in the existing bus definition

To define all elements and attributes supported when defining a bus.

Indicates the maximum number of masters this bus supports. If this element is not present, the number of masters allowed is unbounded.

Indicates the maximum number of slaves this bus supports. If the element is not present, the number of slaves allowed is unbounded.

Indicates the list of system group names that are defined for this bus definition.

Indicates the name of a system group defined for this bus definition.

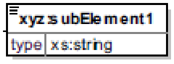
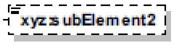
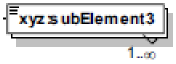
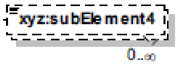
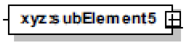
Full description string, typically for documentation

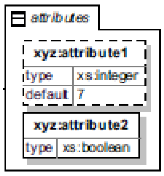
Container for vendor specific extensions.

Figura 14 - *XML Schema* de *busDefinition* [5]

Cada elemento é caraterizado através do seu tipo. Como foi supracitado, os elementos de tipos básicos são representados no diagrama de classes como variáveis, enquanto os elementos de tipo composto são representados através de uma classe. A Tabela 1 resume a conversão dos elementos do *XML Schema* para o diagrama de classes correspondente.

Tabela 1 - Relação entre XML Schema e o Diagrama UML

XML Schema	**Nome do elemento do Schema**	**Diagrama UML**	**Descrição**
	Elemento simples Obrigatório	Variável	O tipo do elemento define o tipo da variável na classe a que pertence. O seu preenchimento é obrigatório.
	Sequencia	-	Não tem representação no diagrama de classes. Define um conjunto de elementos que pertencem a uma classe
	Elemento simples opcional	Variável	O tipo do elemento define o tipo da variável na classe. O seu preenchimento não é obrigatório
	Lista de elementos obrigatória	Lista ligada na classe superior	O tipo do elemento define o tipo da lista ligada a declarar na classe a que pertence. A lista tem de conter no mínimo um elemento
	Lista de elementos opcional	Lista ligada na classe superior	O tipo do elemento define o tipo da lista ligada a declarar na classe a que pertence. A lista pode estar vazia.
	Elemento composto obrigatório	Classe	Um elemento composto contém outros elementos compostos e elementos simples que representam as variáveis da classe.

	Escolha	-	Não tem representação no diagrama de classes. No entanto, obriga a que apenas um tipo dos n elementos que constituem o bloco estejam definidos na classe a que pertencem.
	Grupo de elementos	-	Não tem representação no diagrama de classes e não implicação qualquer tipo de verificação adicional aos elementos que o constituem.
	Grupo de atributos	Variáveis correspondentes aos elementos dentro deste grupo	Tal como os elementos simples, dão origem a variáveis. No entanto estas variáveis têm um tratamento diferente no momento de leitura e escrita dos ficheiros IP_XACT

Dependendo do tipo e da repetibilidade do elemento existem algumas verificações que têm de ser executadas. Tomando como exemplo o elemento *maxMasters*, podemos verificar que o tipo deste elemento é *nonNegativeInteger*, isto é, este elemento aceita valores inteiros numa gama que vai de 0 a +∞. Para além deste tipo, existem outros que merecem especial atenção e para os quais é necessário implementar mecanismos de verificação para que a validação dos documentos seja corretamente executada. Por exemplo o tipo *Name* representa uma *string* (conjunto de carateres) que não pode conter espaços. Outro exemplo é o tipo *spiritURI*, que representa um caminho, absoluto ou relativo, para um ficheiro ou diretório, representado no formato URI. A *framework* deve fornecer suporte ao utilizador nestes casos para evitar erros de preenchimento. Estes mecanismos são depois apresentados e explicados na secção referente ao módulo do Ambiente Gráfico. Na Tabela 2 são apresentados os tipos dos dados que constituem o IP-XACT e as validações que implicam na *framework.*

Tabela 2 - Tipos dos elementos IP-XACT

Tipo	**Descrição**
Boolean	Aceita dois valores: true ou false
configurableDouble	Número decimal de 64 bits
Float	Número decimal de 32 bits
ID e IDREF	Identificador único no documento onde é referido. Começa sempre por uma letra ou _. Não pode conter espaços
instancePath	String com um conjunto de Names separados por uma barra do tipo /.
Integer	Número inteiro
Name, portName	Série de quaisquer caracteres sem espaços. Tem de começar por uma letra, um _ ou :.
NMTOKEN	Série de quaisquer caracteres sem espaços.
NMTOKENS	Série de quaisquer carateres incluindo espaços.
nonNegativeInteger	Números inteiros maiores ou iguais a zero.
positiveInteger	Números inteiros maiores que zero.
scaledInteger	Números inteiros num dos seguintes formatos: 1. Número inteiro na base decimal

	2. Número hexadecimal 3. Número com um sufixo do tipo K,M,G, T
scaledNonNegativeInteger	scaledInteger maior ou igual a zero
scaledPositiveInteger	scaledInteger maior que zero
spiritURI	String representativo de um caminho, absoluto ou relativo, para um ficheiro, executável ou diretoria no formato URI
String	Série de caracteres com espaços
Token	Série de quaisquer caracteres excepto, \n, \t ou \r. Qualquer sequencia de dois ou mais espaços consecutivos são reduzidos a um espaço.

Para aumentar a tolerância a falhas e a independência entre cada módulo, todos os módulos serão responsáveis por fazer verificações formais aos dados trocados. Esta opção implica uma *framework* um pouco mais lenta no final da implementação, mas aumenta a sua robustez, através do aumento dos locais onde são realizadas as verificações. Assim cada classe terá implementado funções para a verificação dos dados que lhe são passados, e caso os dados recebidos não respeitem as especificações requeridas, são descartados ficando os campos preenchidos com os valores por defeito.

A estrutura base que as classes para representação do IP-XACT devem seguir, é ilustrada pela Figura 15.

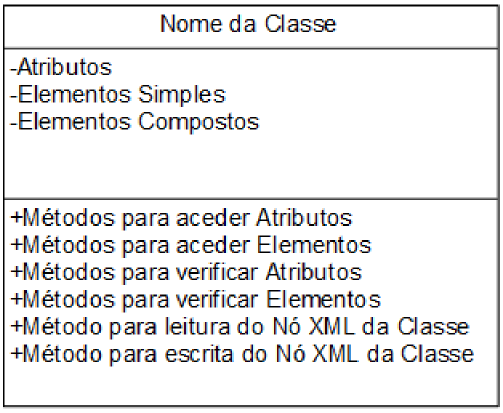


Figura 15 - Estrutura das Classes

O diagrama de classes apresentado na Figura 17 foi obtido através da análise do *XML Schema* do *top element abstractionDefinitio*. Os métodos das classes não estão presentes na imagem porque não era possível obter uma imagem legível com todos os métodos especificados. No entanto, na Figura 16 apresenta-se, a título de exemplo, a classe *abstractDefinition* completa.

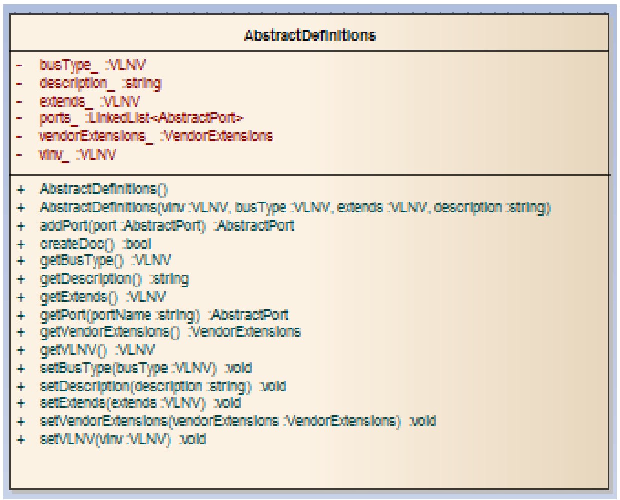


Figura 16 - Classe *abstractDefinition*

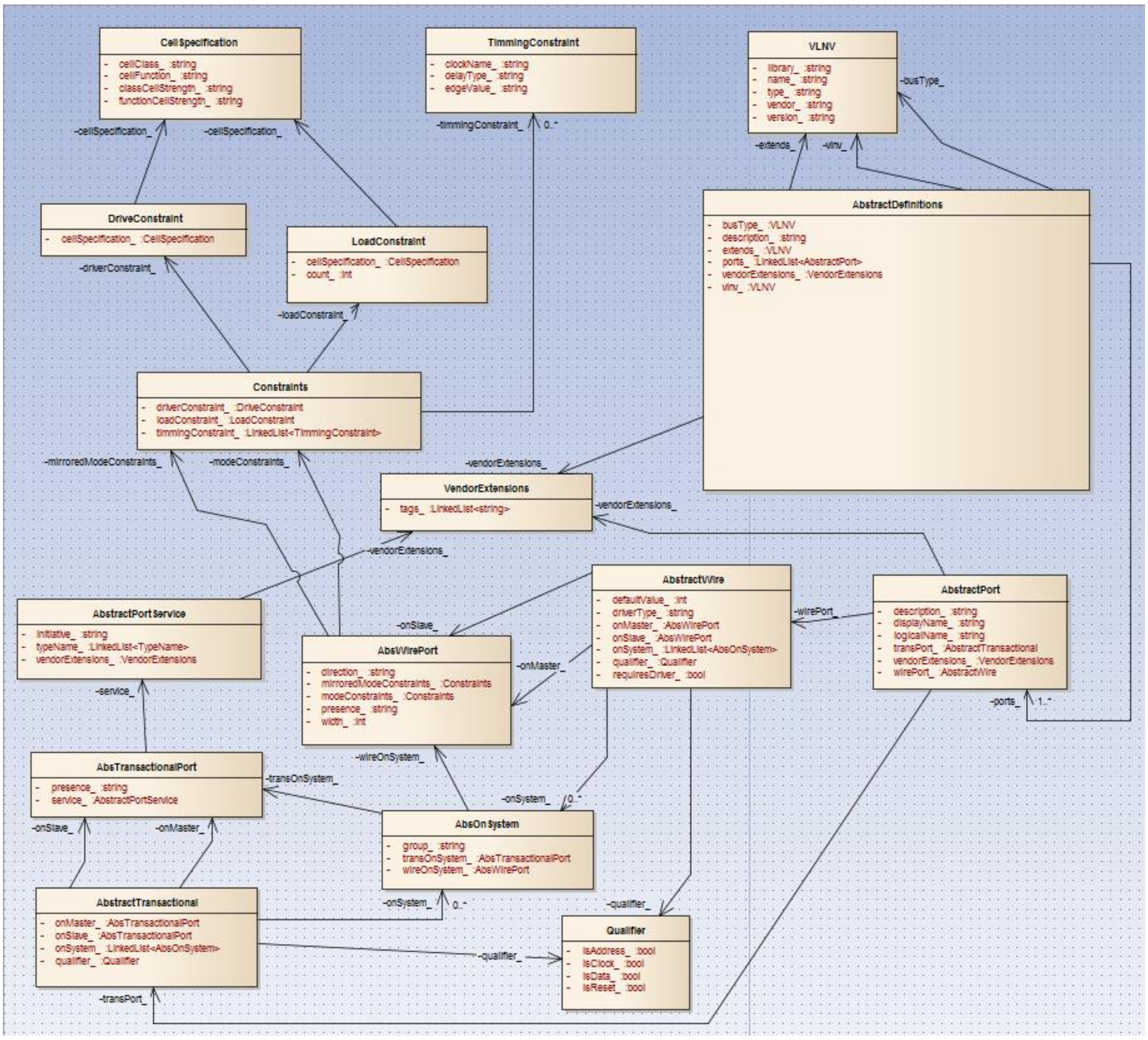


Figura 17 - Diagrama de Classes de *abstractionDefinition*

3.1.2. Ambiente Gráfico (GUI)

No capitulo 2.1 foi apresentadada uma descrição ao *standard* IP-XACT, explicando-se o seu objetivo e particularidades. Uma vez que o projeto proposto consiste na implementação de um ambiente de desenvolvimento compativel com este *standard* é importante realizar-se uma análise aos requisitos impostos pelo mesmo. A Figura 18 foi retirada do documento do IEEE relativo ao IP-XACT, que pode ser consultado em [5], e apresenta uma visão global da estrutura que deve ser respeitada no construção de um ambiente integrado de desenvolvimento baseado no *standard*.

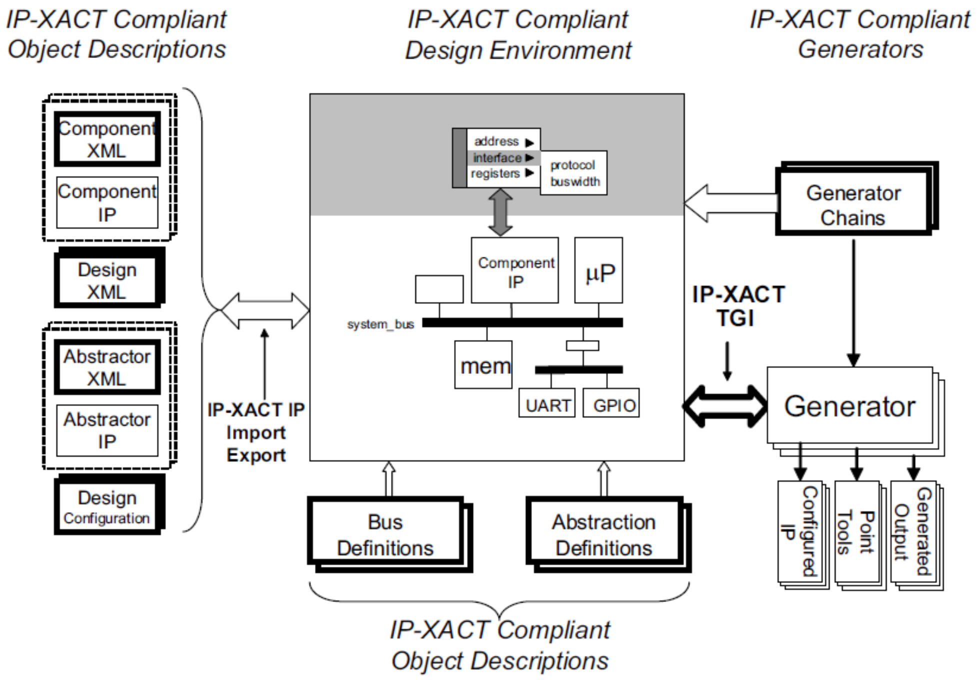


Figura 18 - *Overview* da Interface gráfica (retirado de [5])

Através da análise da Figura 18 é possível identificar os *inputs* e *outputs* para o ambiente de desenvolvimento a implementar, definindo assim as restrições a serem respeitadas. O acesso ao repositório é a principal interface a assegurar. O ambiente de desenvolvimento tem de ser capaz de carregar IPs descritos no formato IP-XACT, ao mesmo tempo que assegura a geração de componentes compatíveis com o *standard*, para serem utilizados em outros projetos. Para além da leitura e escrita para o repositório de IPs do tipo *component*, o acesso às definições dos barramentos, representados na imagem pelos blocos de *Bus Definition* e *Abstraction Definitions*, deve também ser garantido para que todos os processos de validação de conexões possa ser efetuado. Por fim, no que ao repositório diz respeito, o ambiente de desenvolvimento deve permitir carregar um *design* completo e no momento de gravação para além do ficheiro do IP de *design*, devem ser criados os ficheiros referentes ao IP *designConfiruration* e ao IP *component* do respetivo *design*. A interface entre o ambiente de desenvolvimento e os geradores, representados na Figura 18 através do bloco *Generator*, é realizada através da interface TGI (*Tight Generator Interface*). Esta interface consiste num conjunto de mensagens baseadas no protocolo SOAP e os seus detalhes podem ser consultados no documento disponibilizado pelo IEEE relativo ao IP-XACT [5].

Antes de definir quais as funcionalidades a suportar, é importante salientar que existem duas possibilidades no momento de criação de um novo IP. O utilizador por optar por cria um IP *from the scratch*, ou então, construir um novo IP através de outros, já existentes no repositório. No primeiro caso, a análise realizada no capítulo 3.1.1 é suficiente para suportar todas as operações a executar, sendo apenas necessário estruturar o aspeto do ambiente gráfico. Por outro lado, no segundo caso é necessário a implementação de um conjunto de classes para suportar o processo de construção de um *design* através de mecanismos gráficos. Na Figura 19 são apresentadas as possibilidades que devem estar ao dispor do utilizador durante a utilização do ambiente gráfico no caso da criação de um novo *design*.

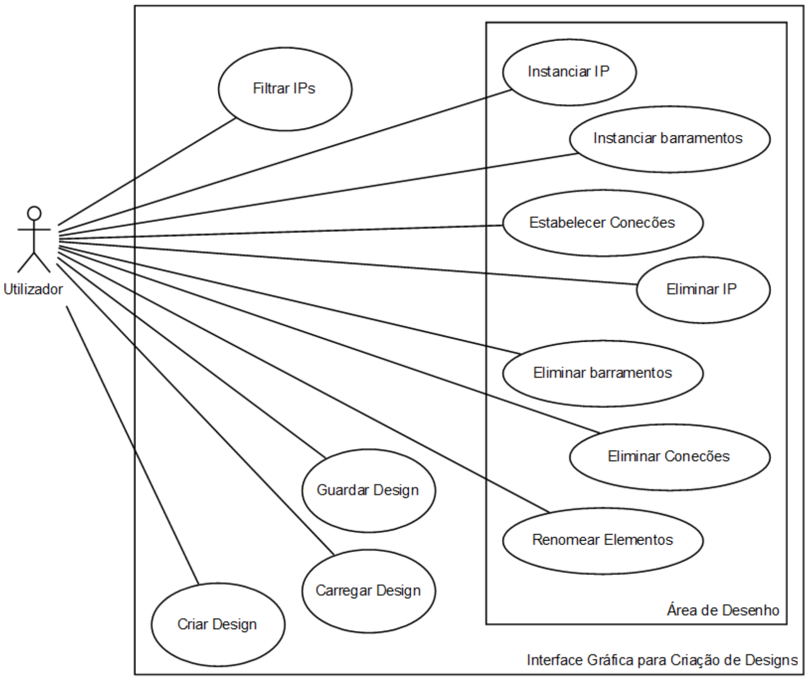


Figura 19 - Casos de uso para a Interface Gráfica

A interface para a criação de um projeto deve apresentar todos os IPs disponíveis para o utilizador. Como supracitado, no capítulo 3.1.1, processo de inicialização do programa consiste na validação do repositório e eliminação de qualquer IP que não seja validado pelo *standard*, portanto assume-se que todos os IPs presentes no repositório são válidos, não sendo necessárias quaisquer verificações adicionais no momento de preenchimentos dos IPs disponíveis. Caso o repositório seja extenso, encontrar o IP desejado pode tornar-se um trabalho moroso, por isso, o repositório deve permitir filtrar IP através dos quatro campos que constituem o ID de cada componente, a VLNV (Vendor, Library, Name, Version).

O utilizador deve ainda ter a possibilidade de criar, guardar ou carregar um projeto, sendo que o processo de criação e atualização de um projeto deve criar, não só um novo *design*, como um novo *component* e uma nova *designConfiguration*. Por fim, dentro da área de desenho, deve ser dada ao utilizador a possibilidade de instanciar e eliminar componentes e barramentos e criar e eliminar conexões entre componentes.

Grande parte dos ambientes de desenvolvimento abordados no Anexo A tinham o objetivo de facilitar o processo de desenvolvimento de código e portanto, funções como coloração de palavras reservadas e completação de código representam uma mais-valia. No entanto, o ambiente proposto pretende fornecer ao utilizador um nível de abstração superior, evitando ao máximo que o utilizador tenha a necessidade de desenvolver código de raiz. A interface que se pretende obter assenta numa perspetiva *drag-and-drop* com validação das operações e *design time*, ou seja, o utilizador deverá ser capaz de instanciar novos IPs através de um simples arrastar do nome do componente para dentro da área de desenho e as validações das conexões entre componentes devem ser feitas no momento em que se seleciona o segundo extremo da conexão. A conexão só deverá ser executada se passar nos testes de validação.

Os principais elementos são a área de desenho, os componentes e os barramentos, sendo que dentro dos barramentos estes podem ser barramentos pertencentes aos componentes instanciados ou então, barramentos instanciados para constituírem a interface do projeto que se pretende criar.

Na classe responsável por representar a área de desenho devem ser implementados todos os métodos relacionados com a gestão do projeto como por exemplo a deteção de sobreposição e relocalização de componentes, o desenho das conexões e os processos de atualização da área de desenho sempre que são efetuados operações de movimentação de componentes ou de *scroll*.

As classes para representação dos componentes e barramentos para além de responsáveis por detetar as operações de clique e de arrasto, são também responsáveis por implementar os métodos das operações de alteração de nome e carregamento das especificações presentes no ficheiro IP-XACT do IP a instanciar. A Figura 20 ilustra o diagrama de classes para a área de desenho da interface gráfica.

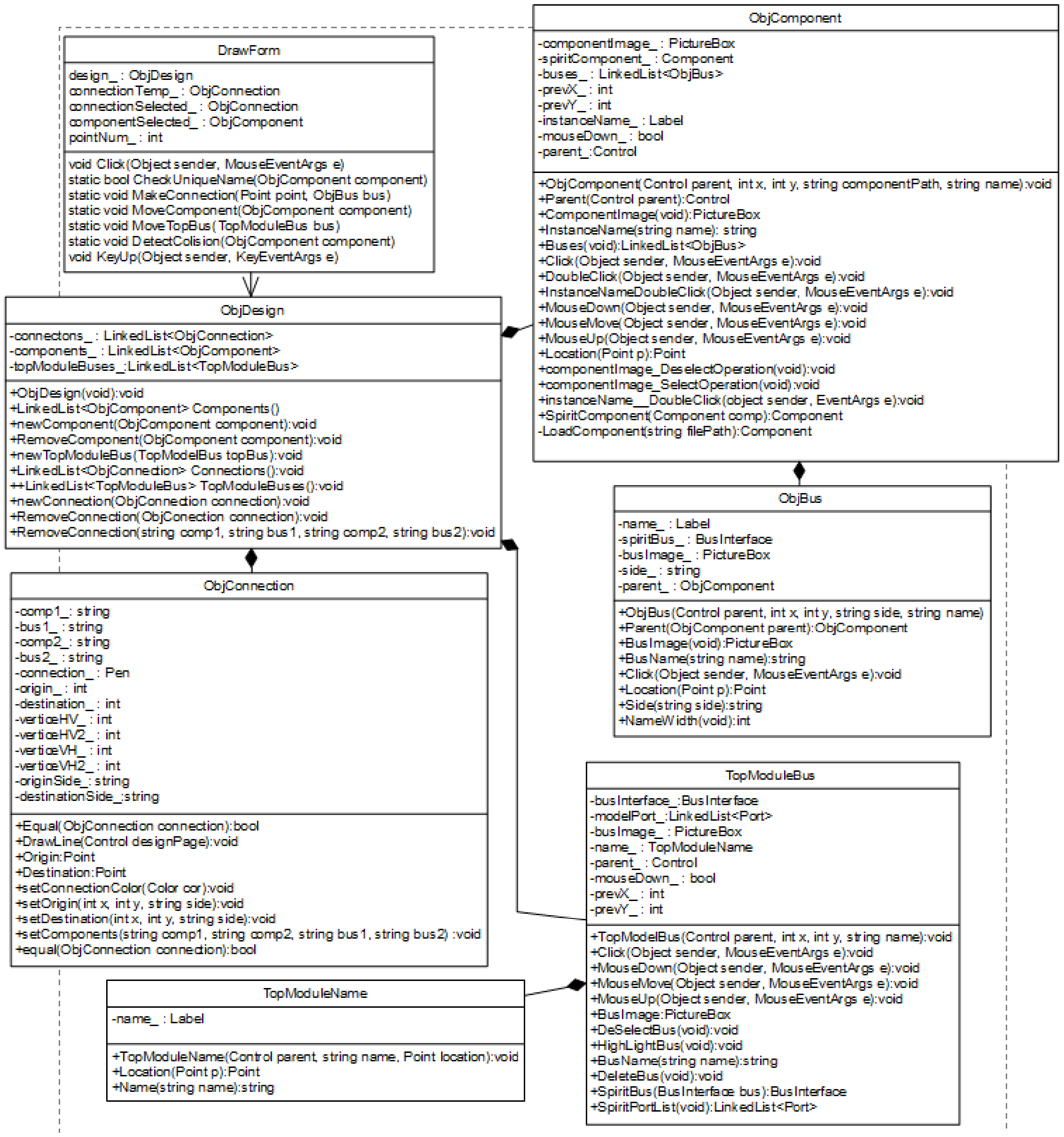


Figura 20 - Diagrama de classes para a área de desenho

Uma vez descritas as funcionalidades e a estrutura base para a interface gráfica importa definir a sequência de inicialização e de fecho. Como supracitado, o utilizador tem a possibilidade de escolher quais os componentes que pretende instanciar no *design* através de uma lista que segue uma estrutura em árvore. Desta forma, o primeiro processo a executar na inicialização do ambiente de desenvolvimento de projetos consiste na criação desta lista. Nesta fase não é necessário exercer qualquer tipo de verificação, uma vez que o repositório já foi validado pelo módulo de gestão do repositório. Por este motivo, a sequência de inicialização deste módulo passa simplesmente por percorrer a árvore de diretórios do repositório para construção da lista com os componentes disponíveis para instanciação. A sequência de fecho do módulo deve considerar todas as alterações realizadas na área de desenho e caso exista alguma modificação em relação ao modelo inicial deve ser apresentada uma mensagem por forma a gravar as alterações efetuadas. Da mesma forma que a opção de carregamento de um projeto já existente deve limpar a área de desenho antes de o carregar. Durante a sequência de fecho a área de desenho deve ser limpa imediatamente antes do fecho da janela.

No que diz respeito à área de desenho, para além dos casos de uso apresentados, deve ser implementadas um conjunto de funcionalidades para facilitar o processo de criação e navegação dos projetos. A área de desenho deve fazer o *highlight* do elemento que está seleccionado. Caso exista um elemento seleccionado, deve ser possivel através da tecla de “del”, apagar o elemento selecionado ou, atarvés da tecla de “esc”, desselecionar o elemento. Deve ser possível mover todos os componentes instanciados e todos os portos pertencentes ao componente de topo, permitindo que o utilizador organize o projeto como desejar. Caso aconteçam casos de sobreposição, a área de desenho deve automaticamente reposicionar o elemento movido, numa área onde não se verifique esta situação. Os mecanismos de *scroll* vertical e horizontal também devem estar presentes, para que a área de desenho não esteja limitada ao tamanho visualizado no ecrã.

Ainda com o objetivo de facilitar o desenvolvimento e navegação do projeto, o ambiente de desenvolvimento deve apresentar listas com os componentes já instanciados e as conexões realizadas. A informação presente nestas listas deve estar constantemente sincronizada com o cenário existente na área de desenho. Deve ainda ser possivel seleccionar os componentes e as conexões a apartir destas listas e adicionar e remover elementos do projeto. Ou seja, todas as operações passiveis de serem realizadas na área de desenho devem também ser possiveis a partir destas listas, com excepção à movimentação de componentes.

3.1.3. Gerador de Código

O módulo de geração de código é responsável pela geração de ficheiros em HDL, e ficheiros que implementam as restrições de *hardware*, necessários para o processo de geração do *bitstream*, através dos ficheiros IP-XACT presentes no repositório. O gerador de código deve também gerar os ficheiros *batch*, com os comandos para invocação das ferramentas externas.

Em suma, este módulo deve criar os ficheiros HDL, caso não existam e criar e executar os *batchs files* para geração dos *bitstreams* e programação das FPGA.

Por forma a potencializar os conhecimentos obtidos em Unidades Curriculares dos anos anteriores, optou-se por dar suporte à linguagem de programação de *hardware* Verilog. Assim, o gerador de código deve permitir a transformação de ficheiros IP-XACT em ficheiros Verilog. Escolhida a linguagem a que se dará suporte, é necessário associar os diferentes elementos presentes no *standard* IP-XACT e o respetivo resultado após convertido para Verilog. A associação realizada é apresentada na Tabela 3.

Tabela 3 - Associação entre elementos IP-XACT e Verilog

Elemento IP-XACT	**Excerto de Verilog associado**
<spirit:componente> <spirit:name>	*module valor_de_spirit:name();*
<spirit:port> <spirit:name> <spirit:wire> <spirit:direction>	*valor_de_spirit:direction valor_de_spirit:name*;
<spirit:wire> <spirit:direction> <spirit:vector> <spirit:left> <spirit:right>	*valor_de_spirit:direction* [*valor_de_spirit:left*:*valor_de_spirit:right*] *valor_de_spirit:name*;
<spirit:*design*> <spirit:componentInstances>	Instanciação de um módulo dentro de outro módulo, em que o tipo do módulo é dado pelo campo *spirit:name* do componente da qual *spirit:componentInstance* é cópia:

<spirit:componentInstance> <spirit:instanceName>	Tipo_do_módulo *valor_de_spirit:instanceName()*;
<spirit:*design*> <spirit:hierConnections> <spirit: hierConnection> <spirit:interface>	Ligação entre o módulo hierárquico superior (*Top Level Module*) e o módulo representado no campo *spirit:interface*: .nome_do_porto_do_submódulo(nome_do_porto_do_módulo_top)
<spirit:*design*> <spirit:interconnections> <spirit: interconnection> <spirit:activeInterface> <spirit:activeInterface>	Ligação entre dois módulos instanciados dentro do módulo hierarquicamente superior. Dá origem a um *wire* para estabelecer a conexão entre os dois *spirit*:*activeInterface*

Para proceder à transformação dos ficheiros IP-XACT em Verilog pode-se utilizar a estrutura de classes que representa o *standard* IP-XACT. No entanto este método torna o processo bastante dependente do ambiente de desenvolvimento e da forma como este foi implementado. Dado o *standard* ser representado à custa de ficheiros XML, é possível a utilização do XSLT (*eXtensible Markup Language Transformation*) para interpretar os ficheiros XML e convertê-los noutro formato, como por exemplo ficheiros de texto ou HTML. Esta abordagem permite desenvolver um gerador de código que depende apenas do *standard* em causa, sendo completamente independente da aplicação e da forma como está desenvolvida a sua arquitetura.

Os ficheiros XSL são bastante utilizados para definir e filtrar os dados e a forma como são apresentados em páginas HTML, permitindo a partir da mesma *stylesheet* criar páginas diferentes caso o ficheiro XML seja alterado. Outra vantagem da utilização de XSLT prende-se ao facto de não ser necessário alterar o programa caso a estrutura do XML seja alterada. Ou seja, se ao ficheiro XML for acrescentado um elemento diferente daqueles que já existiam, só é necessário alterar a *stylesheet* que é aplicada ao XML, mantendo o código da framework intocável. O facto de tornar a transformação independente da aplicação e do ambiente de desenvolvimento e a flexibilidade oferecida no caso de serem necessárias alterações estruturais nos ficheiros XML, tornam esta abordagem numa boa solução para a implementação do gerador de código.

Na secção seguinte é realizada uma breve introdução ao XSLT para facilitar a compreensão das suas capacidades e limitações no processo de transformação do IP-XACT para ficheiros Verilog.

3.1.3.1. Transformação do IP-XACT usando XSLT

Como já foi referido o XSLT é utilizado para processar e filtrar a informação presente em ficheiros XML e transforma-los noutros ficheiros como ficheiros de texto, HTML e XML. Resumidamente, uma *stylesheet* não é mais que um conjunto de regras para aplicar a um ficheiro XML.

A base dos ficheiros XSLT é o elemento *template*. O elemento *template* específica um conjunto de regras que são aplicadas quando uma condição se verifica, no caso em que se tem o atributo *match* definido, ou quando é invocado, no caso de estar especificado o atributo *name*. A estrutura de um *template* é apresentada no exemplo **(1)**.

<xsl:template match=”/”></xsl:template> (1)

O elemento *for-each* é também bastante útil na construção de *stylesheets*. Este elemento permite iterar sobre um conjunto de elementos que respeitem as regras definidas no atributo *select*. Em **(2)** é apresentado um exemplo da utilização deste elemento.

<xsl:for-each select=”expressão_XPath”></xsl:for-each> (2)

Para filtrar resultados é possível a utilização de operações condicionais através do elemento *if*. No caso de existirem várias condições, em vez da utilização de sucessivos *if’s*, é possível aplicar o elemento *choose*. Caso a condição estipulada no atributo *test* do elemento *if* seja verdadeira, todas as regras dentro dele serão executadas. O elemento *choose* é constituído por um ou mais elementos *when*. Cada elemento *when* pode ser visto como um elemento *if* uma vez que também contém um atributo *test* que, quando verdadeiro, permite a execução das regras presentes no elemento. O elemento *choose* pode ainda conter um elemento *otherwise* que corresponde ao conjunto de regras a ser aplicadas por defeito, caso nenhum dos testes em cada elemento *when* seja verdadeira. Em **(3)** é dado um exemplo de uma possível aplicação do elemento *if*, enquanto em **(4)** é apresentado um exemplo para a utilização do elemento *choose*.

<xsl:if test=”position() != last()”></xsl:if> (3)

<xsl:choose> (4)

<xsl:when><xsl:when>

<xsl:otherwise></xsl:otherwise>

</xsl:choose>

Para que não seja necessário executar uma regra sempre que se queira aceder ao valor de um nó mais que uma vez, o XSLT permite a declaração de variáveis **(5)**. Apesar do nome, este elemento é constante e o seu valor tem de ser definido no momento da sua declaração. O elemento *variable* permite também que outros elementos como *for-each’s* e *if’s* sejam utilizados dentro da sua declaração. Em **(6)** é apresentado um exemplo da utilização de uma *variable* em XSLT.

<xsl:variable name=”var” select=”expressão_XPath”></xsl: variable > (5)

<xsl:value-of select=”$var”></xsl:value-of> (6)

O elemento *value-of* pode ser utilizado para escrever o valor presente num nó, ou atributo de um nó, especificado pelo atributo *select* diretamente no ficheiro de *output* **(6)**, ou então para definir o valor de um elemento *variable* quando utilizado dentro do elemento *variable*.

A definição dos nós e atributos utilizados nas expressões XSLT são realizadas recorrendo a expressões XPath. Uma expressão XPath indica o caminho a seguir até alcançar o elemento XML pretendido. A Tabela 4 apresenta a sintaxe do XPath.

Tabela 4 - Sintaxe XPath (adaptado de [35])

Expressão	Descrição
Nome_do_nó	Seleciona todos os nós com nome igual ao especificado
/	Seleciona a partir do nó raiz
//	Seleciona os nós do nó atual que respeitem a condição definida independentemente do nível de aninhamento
.	Seleciona o nó atual
..	Seleciona o nó pai
@	Seleciona atributos do nó

3.1.3.2. Diagramas

Feita a correspondência entre o IP-XACT e o Verilog, são apresentados e explicados, dois dos principais diagramas de fluxo a utilizar na implementação das *stylesheets* para transformação do *standard*. Os restantes diagramas de fluxo estão disponíveis no Anexo C.

Pela observação das correspondências da Tabela 3, a implementação do ficheiro responsável pela transformação de um ficheiro IP-XACT de um componente para código Verilog é uma tarefa relativamente simples. Na Figura 21 é possível visualizar o diagrama de fluxo responsável por esta transformação.

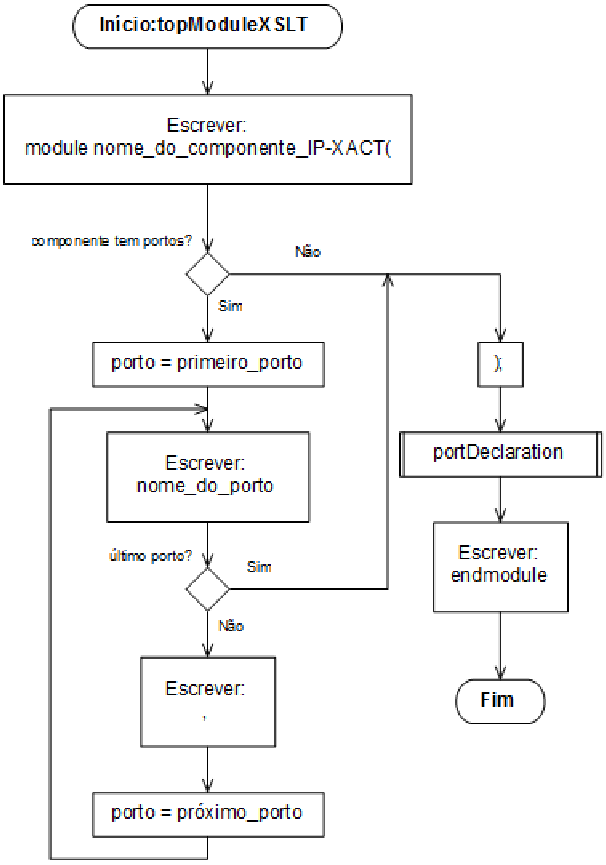


Figura 21 - Fluxograma de conversão de componente IP-XACT para Verilog

A operação de conversão é iniciada com a escrita da palavra reservada *module* seguida do nome do componente que corresponde ao valor do elemento *spirit:name* dentro de *spirit:componente*. De seguida, caso o componente tenha elementos do tipo *spirit:port* é executado um ciclo para escrever todos os portos que constituem o componente. Após escritos todos os portos, passa-se ao processo de declaração de cada um dos portos. Este processo é executado pela função *portDeclaration*, cujo fluxograma se encontra disponível no Anexo C. Em *portDeclaration* volta-se a percorrer todos os portos do componente, mas desta vez, para além do nome do porto são também analisados os elementos *spirit:direction*, para saber o tipo do porto e os elementos *spirit:left* e *spirit:right*, presentes em *spirit:vector*, para determinar o número de bits de cada sinal. Por fim escreve-se a palavra reservada *endmodule* para terminar o módulo.

No caso de *designs* IP-XACT o processo de transformação torna-se bastante mais complexo. Os ficheiros de um *design* IP-XACT podem ser interpretados como grafos onde os elementos do tipo *spirit:componentInstance* representam os vértices do grafo e os elementos do tipo *spirit:interconnection* as arestas. Analisando estes ficheiros, chega-se à conclusão que não existe uma ordem específica para a representação das ligações no documento IP-XACT. No entanto no processo de transformação essa ordem pode ser determinante para que a conversão seja realizada corretamente. Para que se perceba a importância desta questão é apresentado um exemplo ilustrativo.

Imagine-se a existência de um ficheiro de um projeto com quatro componentes cada um deles com um barramento de dados do tipo *inout* com o nome *data*, ligados entre si. A Figura 22 ilustra esta situação.

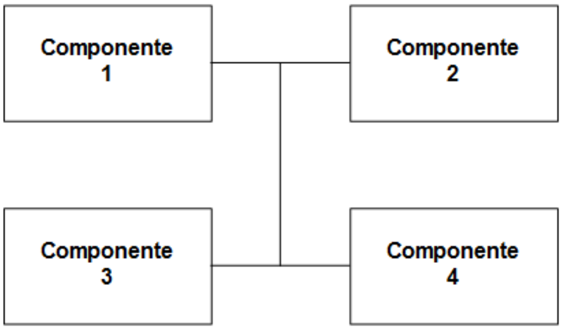


Figura 22 - Exemplo de *design* IP-XACT

Uma possível representação das interligações no ficheiro IP-XACT seria a demonstrada na Figura 23.

```
<spirit:interconnections>
    <spirit:interconnection>
    <spirit:name>comp1_data_to_comp2_data</spirit:name>
    <spirit:activeInterface spirit:componentRef="comp1"
spirit:busRef="data"/>
    <spirit:activeInterface spirit:componentRef="comp2"
spirit:busRef="data"/>
    </spirit:interconnection>
    <spirit:interconnection>
    <spirit:name>comp3_data_to_comp4_data</spirit:name>
    <spirit:activeInterface spirit:componentRef="comp3"
spirit:busRef="data"/>
    <spirit:activeInterface spirit:componentRef="comp4"
spirit:busRef="data"/>
    </spirit:interconnection>
    <spirit:interconnection>
    <spirit:name>comp2_data_to_comp3_data</spirit:name>
    <spirit:activeInterface spirit:componentRef="comp2"
spirit:busRef="data"/>
    <spirit:activeInterface spirit:componentRef="comp3"
spirit:busRef="data"/>
    </spirit:interconnection>
</spirit:interconnections>
```

Figura 23 - XML de exemplo das interligações num *design* IP-XACT

Analisando a Figura 22 rapidamente se chega à conclusão de que todos os quatro componentes se encontram ligados, no entanto, no ficheiro XML só se consegue obter essa informação relacionando as três ligações, ou seja, sabe-se que o componente 1 está ligado ao componente 4 porque o componente 3 está ligado ao componente 2, que também está ligado ao componente 1 através do mesmo barramento. Portanto, torna-se necessário percorrer o ficheiro mais que uma vez para determinar todas as possíveis ligações em comum entre todos os componentes, para que não se criem mais portos auxiliares do que aqueles que são necessários. Devido às limitações das *stylesheets*, o processo torna-se complicado, chegando mesmo a ser impossível de resolver o problema utilizando apenas a versão 1.0 do XSLT. Como já foi referido, as variáveis em XSLT são constantes e o seu valor não pode ser alterados após ter sido definido. Para resolver este problema é necessário utilizar a versão 2.0 do XSLT, que disponibiliza os elementos *function* e *sequence*. Estes elementos permitem correr os ficheiros XML de forma recursiva, guardando os valores de retorno de cada invocação. Este procedimento é ilustrado pelo fluxograma representado pela Figura 24 e Figura 25.

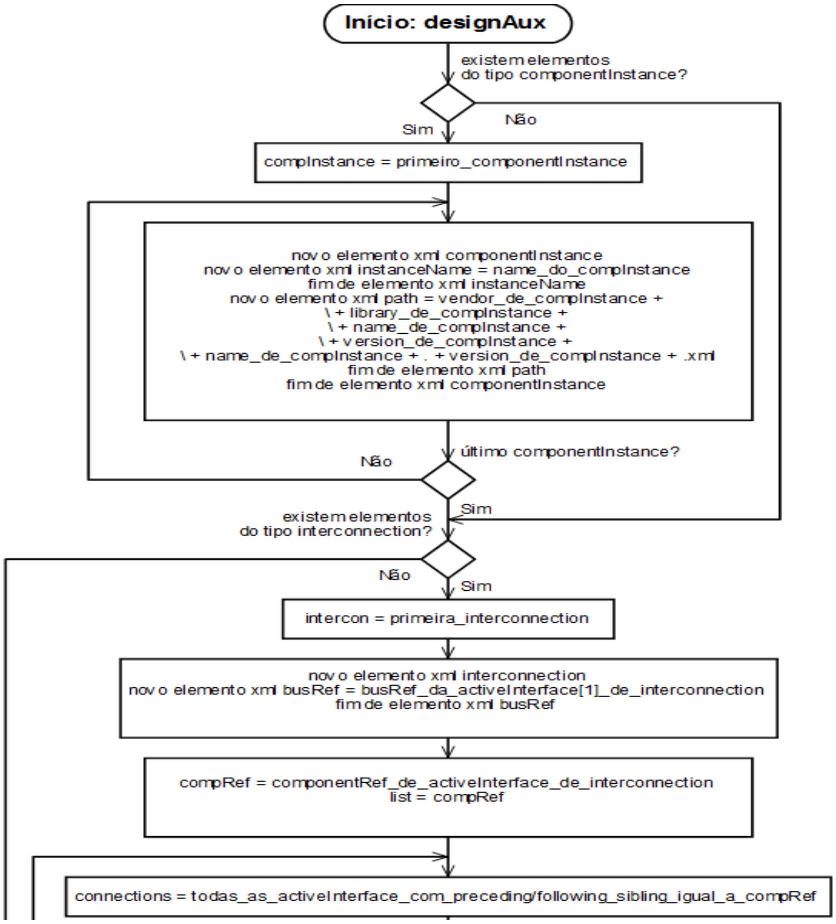


Figura 24 - Fluxograma para geração de ficheiro XML auxiliar de *design* parte 1

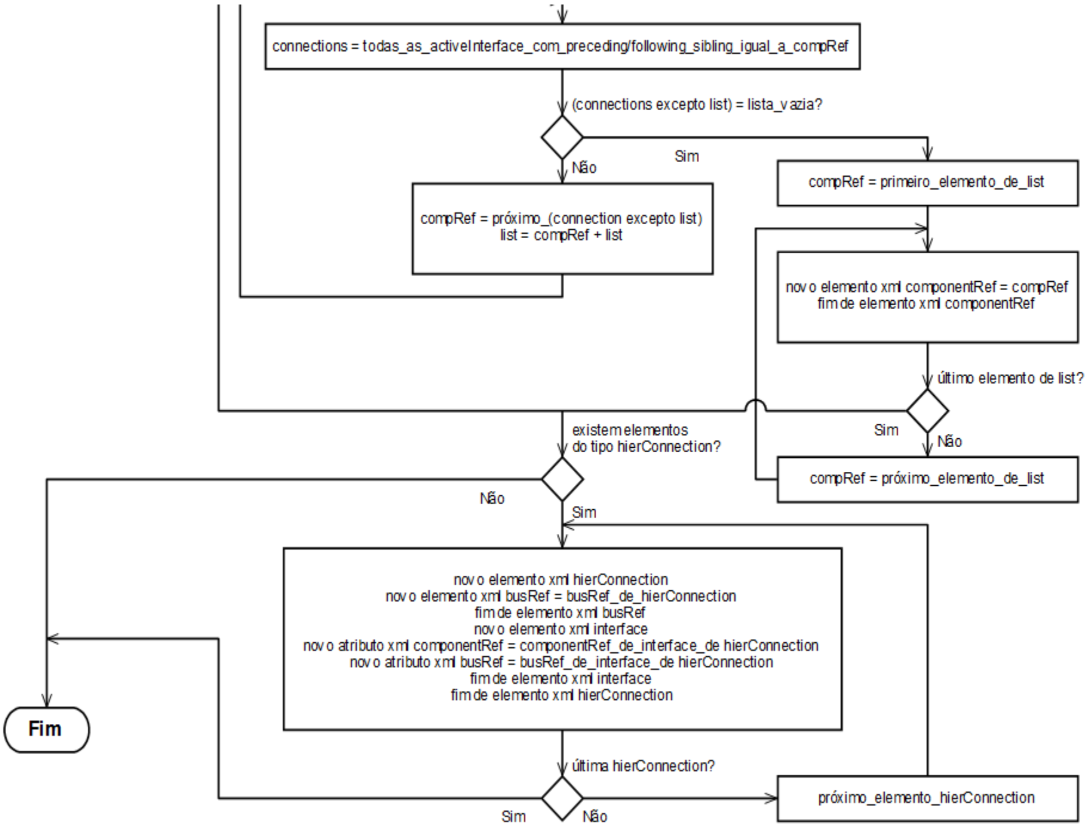


Figura 25 - Fluxograma para geração de ficheiro XML auxiliar de *design* parte 2

Devido à complexidade do problema, fazer a transformação direta de um *design* IP-XACT para Verilog implicaria um grande conjunto de ciclos encadeados e de difícil compreensão. Para simplificar o problema optou-se pela divisão do processo de transformação em duas etapas, suportadas por dois ficheiros XML auxiliares. Analisando o fluxograma anteriormente apresentado pode-se constatar que o ciclo de procura das ligações será executado por cada *spirit:activeInterface* presente nas *spirit:interconnection* do *design*. Isto implica que no limite, a situação em que só existem dois componentes conectados, ou seja, só existe uma *spirit:interconnection*, pelo menos uma repetição vai ocorrer. Desta forma a primeira *stylesheet* é responsável por copiar todos os elementos do tipo *spirit:componentInstance* e *spirit:hierConnections* e por determinar todas as ligações entre componentes, o que implica um conjunto de *spirit:interconnetion* repetidas. A segunda *stylesheet* irá atuar sobre este ficheiro XML com repetições a fim de gerar um ficheiro XML final sem repetições. O fluxograma para a *stylesheet* está disponível no Anexo C. Mais uma vez, a *stylesheet* limita-se a copiar os elementos do tipo *spirit:componentInstance* e *spirit:hierConnections*. Se o elemento for do tipo *spirit:interconnection* é realizado um ciclo para determinar a unicidade do elemento recorrendo aos seus *siblings* precedentes. Caso seja único, o elemento é escrito no ficheiro XML, caso contrário nenhuma ação é efetuada. Assim, no final da geração do segundo XML, tem-se um ficheiro com todos os dados fundamentais para a geração de um ficheiro Verilog representativo do *design* IP-XACT, sem quaisquer repetições. Pegando novamente no exemplo da Figura 22, o ficheiro XML esperado será algo semelhante ao ilustrado na Figura 26.

```
<interconnections>
    <interconnection>
        <busRef>data</busRef>
        <componentRef>comp1</componentRef>
        <componentRef>comp2</componentRef>
        <componentRef>comp3</componentRef>
        <componentRef>comp4</componentRef>
    </interconnection>
</interconnections>
```

Figura 26 - XML auxiliar de *design* IP-XACT sem repetições

Por fim, o processo de transformação do *design* IP-XACT termina com a aplicação de uma *stylesheet* ao ficheiro XML sem repetições. Esta *stylesheet* gera um ficheiro Verilog recorrendo ao segundo ficheiro de *design* auxiliar e aos ficheiros componente IP-XACT do módulo hierarquicamente superior e dos componentes instanciados no *design* através do elemento *spirit:componentInstance*.

Terminada a transformação, o próximo passo é a geração do *bitstream* e a programação do dispositivo alvo. O processo de geração do *bitstream* é totalmente dependente da plataforma alvo e depende do distribuidor da plataforma, pelo que a utilização das ferramentas fornecidas pelos distribuidores é essencial. Para que não exista a necessidade de aprender a trabalhar com mais do que uma ferramenta, a *framework* a desenvolver deve realizar a interface entre o utilizador e as ferramentas de apoio ao desenvolvimento da plataforma que se está a utilizar. Para isto deve invocar as ferramentas através de uma interface baseada na linha de comandos, onde são passados todos os parâmetros necessários à geração do respetivo *bitstream* e programação do dispositivo, ao mesmo tempo que redireciona o *output* destas ferramentas para a consola da *framework*, a fim de garantir ao utilizador um acompanhamento ao longo do processo.

Uma vez que a Xilinx e a Altera cobrem cerca de 80% do mercado de FPGA, a *framework* deve conter algumas das plataformas disponibilizadas por estes dois distribuidores para dar suporte à geração de código e programação. Nesse sentido, uma análise mais detalhada da interface da linha de comandos, das ferramentas disponibilizadas por estas duas entidades, é abordada no subcapítulo 3.3.

Abordados os aspetos importantes relacionados com o projeto a desenvolver é exposto na Figura 27 o diagrama de atividades que a *framework* deve seguir, no caso da criação de um componente IP-XACT. O diagrama de atividades para um *design* IP-XACT é muito semelhante e está disponível no Anexo D.

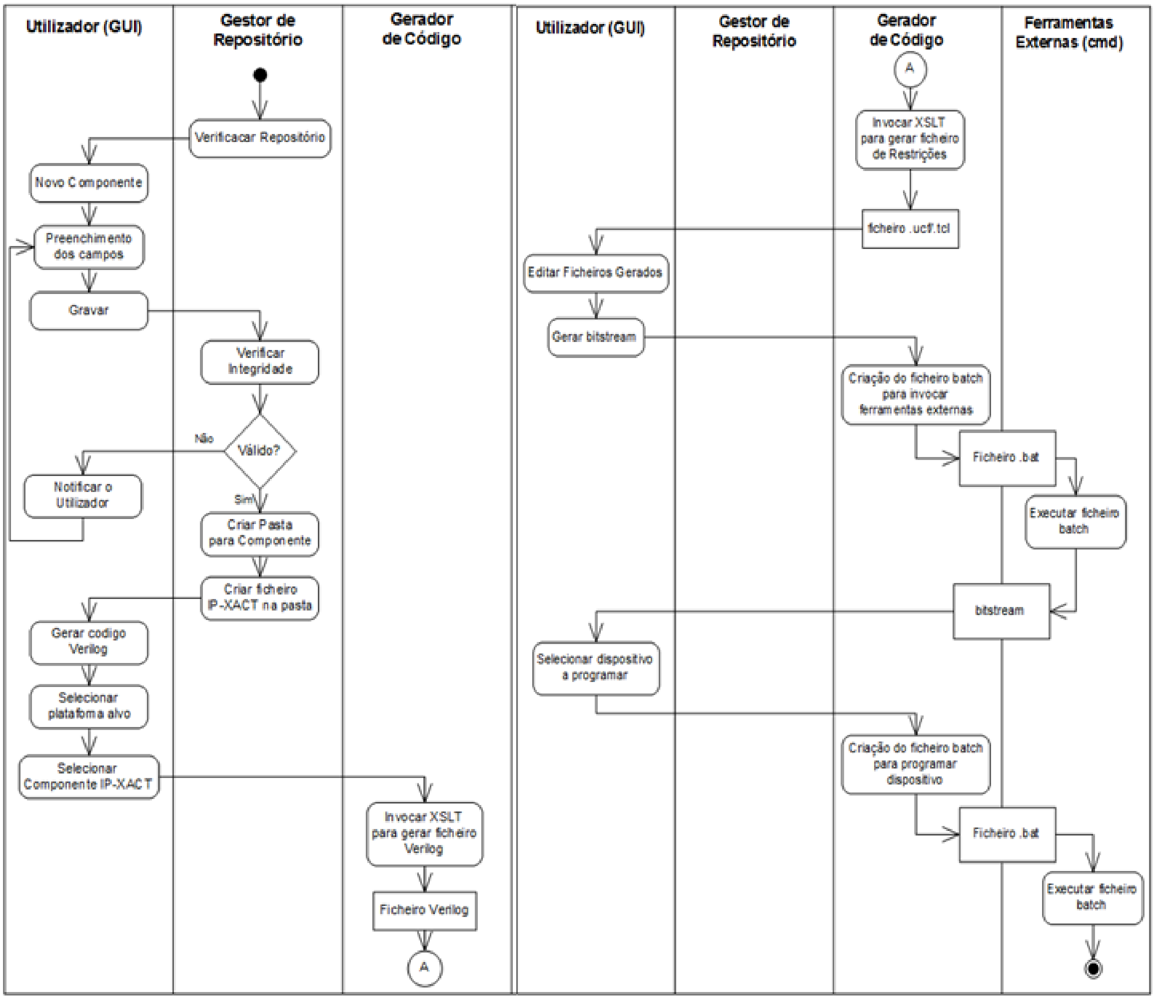


Figura 27 - Diagrama de Atividades

O fluxo normal de execução inicia-se com uma verificação ao repositório para garantir que não existem componentes que não respeitam o IP-XACT. De seguida o utilizador tem a possibilidade de criar um novo componente, assim como editar um componente existente no repositório ou então criar ou editar um *design*. Caso seja selecionado a criação de um novo componente, um painel gráfico com os campos a preencher deve ser apresentado. Concluído o preenchimento de todos os campos relevantes e executada a ordem de gravar ficheiro, a *framework* deve em primeiro lugar verificar se o componente que se pretende gravar cumpre as especificações exigidas pelo *standard*. Caso exista algum erro de preenchimento o utilizador deve ser advertido com uma mensagem de erro. Caso não haja nada de errado, criar-se automaticamente um diretório específico para o ficheiro IP-XACT e de seguida grava-se o ficheiro. O utilizador pode depois exercer uma ordem para geração do código Verilog correspondente ao componente desenvolvido ou outro componente existente no repositório. Selecionado o componente a transformar, a *stylesheet* para conversão de componentes IP-XACT é invocada criando um ficheiro Verilog. Seguidamente é invocada a *stylesheet* responsável pela geração do ficheiro que contém as restrições do componente, como por exemplo o mapeamento dos sinais do ficheiro Verilog com os pinos da plataforma alvo. Após a sua criação, estes dois ficheiros são abertos automaticamente, dando ao utilizador a possibilidade de os editar. Ao fechar estes ficheiros a opção de geração de código fica disponível para o utilizador. Quando executada a ordem de geração de código, a *framework* cria um ficheiro *batch* com os comandos necessários para a geração de código para a plataforma selecionada e executa-o de seguida. O resultado desta atividade é um ficheiro *.bit* que pode depois ser utilizado para programar a FPGA. Por fim, uma lista com os dispositivos possíveis de serem programados é apresentada ao utilizador. Depois de selecionada a FPGA a programar, a *framework* cria outro ficheiro *batch* com os comandos necessários à programação do dispositivo e executa-o, resultando na programação da FPGA e terminando o fluxo de atividades. O Processo pode depois ser repetido a partir da atividade de criação de um novo componente ou a partir da geração do código Verilog de um componente.

3.2. Convenção de Nomes e Localizações

Uma vez que o *standard* IP-XACT não define qualquer tipo de convenção para a estrutura do repositório ou para a nomenclatura dos ficheiros, para manter o repositório organizado e eliminar algumas das limitações do XSLT no momento da geração automática de código, o autor da presente dissertação definiu uma convenção de nomes que garante o correto funcionamento de todo o ambiente de desenvolvimento. A listagem seguinte apresenta as regras definidas pela convenção de nomes criada.

1. Todos os ficheiros IP-XACT estão localizados numa árvore de diretórios que respeita o padrão: *repositor_root/Vendor/Library/Name/Version/*;

2. Os ficheiros gerados automaticamente pelo ambiente de desenvolvimento que não pertençam ao *standard*, como por exemplo ficheiros HDL, estão localizados numa árvore de diretórios que respeita o seguinte padrão: *repositor_root/Vendor/Library/Name/Version/ProjectFiles*;

3. Os nomes dos ficheiros de *component* e *busDefinition* respeitam o padrão: *Name.Version.xml*;

4. O nome dos ficheiros de *abstractionDefinition* respeita o padrão: *Name.absDef.Version.xml*;

5. O nome dos ficheiros de *design* respeita o padrão: *Name.design.Version.xml*;

6. O nome dos ficheiros de *designConfiguration* respeita o padrão: *Name.designcfg.Version.xml*.

A violação desta convenção não impede o correto funcionamento do ambiente de desenvolvimento, mas não garante a correta execução das opções de geração automática de código HDL.

No Anexo E é possível visualizar-se uma breve descrição sobre as duas plataformas de teste utilizadas para a validação da *framework* desenvolvida.

3.3. Ferramentas Externas

Como já referido na presente dissertação, a *framework* que se propõe desenvolver incluirá geradores para os dois principais distribuidores de FPGA, a Xilinx e a Altera. Por forma a realizar uma interface o mais transparente possível para o utilizador com as ferramentas de apoio ao desenvolvimento destes dois distribuidores foi realizado um estudo de cada uma das interfaces a partir da linha de comandos. Desta forma são de seguida apresentadas as análises realizadas a cada uma das ferramentas.

3.3.1. Linha de Comandos da Xilinx

De acordo com o manual do utilizador da linha de comandos da Xilinx [39], o *design flow* divide-se em três fases: síntese; implementação; verificação. Cada uma destas fases envolve uma ou mais ferramentas disponibilizadas pela Xilinx. De seguida são apresentadas as ferramentas assim como a sintaxe para a sua utilização, divididas por cada uma das três fases de *design flow*.

O processo de síntese consiste na conversão dos ficheiros HDL que constituem o projeto, num ficheiro NGC, que corresponde ao *netlist* do projeto. Este processo é da responsabilidade da ferramenta da Xilinx XST (Xilinx Synthesis Technology). Este utilitário pode ser executado de duas formas: linha de comandos do XST; ou através da linha de comandos do PC, correndo um *batch* com o comando para executar o utilitário [39].

Para a presente dissertação interessa explorar o segundo método de execução apresentado. O comando apresentado em **(7)** ilustra a execução da ferramenta XST. Neste comando *-ifn nome.v* representa o ficheiro de *input* para o XST, *-ifmt linguagem* representa a opção que define a linguagem do ficheiro de *input*, as opções possíveis são: Verilog; VHDL; ou mixed. A opção *-ofn nome* determina o nome do ficheiro de *output*, *-p* indica o dispositivo para o qual se está a sintetizar o projeto, *-opt_mode* define o modo de otimização, as opções possíveis são *speed* ou área e por fim, a opção *-opt_level* pode tomar os valores 1 ou 2 [40].

```
run -ifn ficheiro.v -ifmt Linguagem -ofn ficheiro -p dispositivo -opt_mode speed -opt_level 1 | xst
```

(7)

Criado o ficheiro NGC, o processo de implementação tem como objetivo final, obter um *bitstream* para programar a FPGA. Isto é conseguido através das ferramentas NGDBuild, MAP, PAR e BitGen, invocando-os na respetiva ordem [39].

O NGDBuild recebe como entrada um *netlist* no formato NGC ou EDIF e cria um ficheiro NGD (*Native Generic Database*) que contém uma descrição em termos de elementos lógicos, do projeto que se pretende implementar. O formato do comando para a execução da ferramenta NGDBuild encontra-se descrito em **(8)**. A opção *-p* define a placa para a qual se pretende gerar o ficheiro NGD enquanto a opção *-uc* permite indicar o ficheiro com as restrições do projecto, descritas num ficheiro UCF (*User Constraints File*). A assinatura do comando de execução da ferramenta NGDBuild termina com o nome do ficheiro gerado anteriormente pelo comando XST.

```
ngdbuild -p dispositivo -uc ficheiro.ucf ficheiro.ngc
```
(8)

O mapeamento entre o projeto lógico criado pela ferramenta NGDBuild e a FPGA que se pretende programar é realizado através da ferramenta MAP. O resultado do comando de execução da ferramenta MAP **(9)** é um ficheiro NCD (*Native Circuit Description*) que corresponde à representação do projeto mapeado nos componentes da placa da Xilinx que se pretende programar [39]. A opção *-detail* indica à ferramenta que deve gerar um relatório detalhado da operação de map enquanto a opção *-pr* dá indicação que devem ser colocados registos no portos de I/O. A opção *-pr* pode tomar os valores *i*, para colocar registos só nos portos de *input*, *o*, para colocar registos só nos portos de *output* e *b*, para colocar registos em ambos os portos de *input* e *output*.

```
map -detail -pr b ficheiro.ngd
```
(9)

Após a geração do ficheiro NCD executa-se a ferramenta PAR **(10)** para realizar o processo de *place & route*. Este processo gera outro ficheiro NCD que depois será utilizado como *input* do comando de execução da ferramenta BitGen [39]. A opção *-w* dá a indicação para reescrever quaisquer ficheiros existentes com o mesmo nome. O primeiro ficheiro com a extensão NCD corresponde ao ficheiro de *input* que resulta da fase anterior enquanto o segundo ficheiro NCD corresponde ao nome do ficheiro que será gerado pela ferramenta PAR. Por fim o ficheiro com extensão PCF (*Physical Constraints File*) dá as restrições de tempo que devem ser cumpridas no processo de *place & route*. A especificação deste último ficheiro não é estritamente necessária uma vez que a ferramenta PAR consegue deteta-lo automaticamente, caso este contenha o mesmo nome que o primeiro ficheiro NCD.

```
par -w ficheiro.ncd parout.ncd ficheiro.pcf
```
(10)

Por fim, para obter o *bitstream* necessário à programação da FPGA recorre-se ao comando BitGen **(11)**. Esta ferramenta recebe o ficheiro NCD gerado pelo comando de execução da ferramenta PAR e retorna como *output* um ficheiro binário do tipo BIT. Neste exemplo, a opção *-g* permite definir algumas configurações relacionadas com a geração do *bitstream*. Neste caso configurou-se o sinal de relógio com o valor JTAGCLK que indica que o sinal de relógio a ser usado no momento da programação da FPGA é o sinal proveniente do JTAG. Configurou-se também a opção de geração de um CRC (*Cyclic Redundancy Check*) que permite detetar a ocorrência de erros no processo de geração do *bitstream*. Realizadas todas as configurações, é passado ao comando de execução da ferramenta BitGen o ficheiro NCD resultante da operação de *place & route* seguido do nome do ficheiro de *output* a criar. Tal como no caso anterior o comando termina com a passagem do ficheiro PCF como *input* [39].

```
bitgen -w -g StartUpCLK:JTAGCLK -g CRC:Enable parout.ncd
ficheiro.bit ficheiro.pcf
```

(11)

A fase de verificação consiste no teste do projeto implementado para determinar se este cumpre todos os requisitos de tempo definidos e se o seu comportamento corresponde ao esperado [39]. No entanto, esta fase não tem implicações no processo de obtenção do *bitstream* e por esse motivo não se explorou esta vertente das ferramentas oferecidas pela Xilinx.

A secção seguinte aborda os utilitários disponibilizados pela Diligent, que são utilizados para programar algumas das famílias de FPGA comercializadas pela Xilinx.

O programa ADEPT, disponibilizado pela Diligent utiliza um conjunto de utilitários que permitem a programação de algumas famílias de FPGA disponibilizadas pela Xilinx. O ADEPT fornece todo um interface gráfico ao utilizador que permite determinar que dispositivos possíveis de serem programados estão conectados ao computador, assim como programar estes mesmos dispositivos [37]. No entanto, para que a interface entre a *framework* que se propõe desenvolver e o ADEPT seja transparente ao utilizador, optou-se por, em vez de invocar a aplicação, fazer um pedido a um dos utilitários utilizados pelo ADEPT com o nome de *djtgcfg* (*Diligent JTAG Config Utility*). De todos os comandos oferecidos por este utilitário, interessam apenas dois para a *framework* proposta pela presente dissertação. Os comandos de relevância são: o *enum* e o *prog*.

O comando *enum* permite determinar quais as FPGA suportadas pela Diligent que se encontram conectadas ao computador. O comando *prog* possibilita a programação da respetiva FPGA. Juntamente com o comando *prog* devem ser especificadas as opções *-d*, que define o nome do *kit* de desenvolvimento que se pretende programar, *-i* que define se se pretende programar a memória *flash* da FPGA ou então programar diretamente a FPGA e *-f*, para especificar o ficheiro binário a utilizar para programar a FPGA [41]. Em **(12)** e **(13)** são apresentadas as sintaxes para a utilização dos comandos *enum* e *prog* respetivamente.

djtgcfg enum (12)

djtgcfg prog -d dispositivo -i 0 -f programa.bit (13)

3.3.2. Linha de Comandos da Altera

A Altera utiliza o *software* Quartus II para desenvolver os projetos para as suas FPGA. Este *software* está provido com uma ferramenta para cada um dos passos do *design flow* de sistema com base em FPGA [42].

À semelhança do que acontece com o *design flow* da Xilinx, o primeiro passo passa por criar um projeto baseado em ficheiros HDL. Para esse efeito utiliza-se o comando de execução da ferramenta quartus_sh **(14)** com a opção *-t* seguido de um ficheiro TCL. Os ficheiros com uma extensão TCL são os ficheiros utilizados pelo Quartus II para criar um novo projeto. Estes ficheiros contêm todas as restrições a aplicar ao projeto que se pretende desenvolver [43]. Neste ficheiro é especificada a família de FPGA que se pretende utilizar, quais os ficheiros HDL que constituem o projeto e todas as restrições de tempo e de mapeamento entre os portos da FPGA e os sinais presentes nos ficheiros HDL. A ferramneta quartus_sh pode ser invocada com a opção --*flow*, o que permite a compilação de todo o projeto utilizando apenas um comando[42].

quartus_sh –t projeto.tcl (14)

Depois de criado o projeto executa-se o comando de execução da ferramenta quartus_map **(15)** seguido do nome do projeto para sintetizar e mapear o projeto. A ferramenta quartus_map pode também ser utilizada para criar um novo projeto, evitando assim a utilização do comando quartus_sh. Para tal basta acrescentar ao comando as opções --*source*=

seguido do ficheiro HDL que corresponde ao módulo hierarquicamente superior do projeto e --*family*= seguida da família de FPGA que será o alvo do projeto a desenvolver [42][43].

quartus_map projeto (15)

O próximo passo envolve a execução da ferramenta quartus_fit seguido do nome do projeto sobre o qual este deve ser executado. Esta ferramenta permite o *place & route* do projeto e a sua execução pode ser realizada como demonstrado em **(16)**. O comando para a execução da ferramenta quartus_fit pode conter as opções --*part*= seguido do dispositivo para o qual queremos executar o *placement & routing* e pela opção --*pack_register*= que permite definir restrições de *routing* por forma a otimizar o projeto em termos de área ou tempo [43].

quartus_fit projeto (16)

O comando para a execução da ferramenta quartus_asm é utilizado para gerar os ficheiros para programar a FPGA. O comando é executando escrevendo o nome da ferramenta seguido do projeto a que se pretende aplicar a operação **(17)** [42][43].

quartus_asm projeto (17)

Por fim, o comando para a execução da ferramenta quartus_pgm **(18)** permite programar todos os dispositivos da Altera [43]. As opções mais utilizadas são: -*l*, para fazer o display da lista de dispositivos disponíveis; -*c*, para especificar o tipo de cabo a utilizar, normalmente é passada a esta opção o valor *usb-blaster* e -*m*, para definir o modo de programação a utilizar. Os valores aceites por esta opção são JTAG, PS, AS ou SD. A opção -*o*, permite definir as operações a executar sobre o dispositivo no momento de programação. Esta opção pode ser definida com vários valores, entre eles, *p* (programar), *r* (limpar), *v* (verificar) e *b* (blank-check), seguidos de um ponto e vírgula (;), seguindo-se o nome do ficheiro a programar, de um @ e por fim, do índice do dispositivo que se quer programar. Caso só exista um dispositivo a última parte do comando pode ser omissa.

quartus_pgm -c usb-blaster -m jtag -o p;ficheiro.sof (18)

Capítulo 4

IMPLEMENTAÇÃO

Neste capítulo é apresentada a fase de implementação dos três módulos descritos no capítulo 3. O capítulo começa com a apresentação da implementação das classes que representam o *standard* e das funções de suporte à sua gestão como as funções de validação, de leitura e de escrita. De seguida é apresentado o aspeto final do ambiente gráfico criado, juntamente com a implementação de alguns dos mecanismos de interface implementados. A apresentação do GUI encontra-se dividida em duas partes, (i) uma referente à geração de componentes simples (*flat component*), outra (ii) referente à criação de componentes hierárquicos (*hierarchical component*). A implementação do terceiro módulo foi dividida em duas secções distintas. Uma para o gerador de código e outra para a invocação das ferramentas externas. Na secção relativa ao gerador de código são apresentados os ficheiros XSLT criados e os métodos para invocação dos mesmos. O capítulo termina com a secção referente ao invocador de ferramentas externas, onde são apresentados os trechos de código responsáveis por invocar as ferramentas externas e por receber e processar os dados devolvidos.

4.1. Gestor de Repositório

O processo de implementação do gestor de repositório pode ser dividido em 4 fases: (i) implementação das classes base para representação do *standard*; (ii) implementação da função de escrita de um novo elemento IP-XACT; (iii) implementação das funções de leitura de um elemento IP-XACT; (iv) validação dos ficheiros lidos/gerados. Cada uma destas fases é depois subdividida em subfases mais pequenas. Seguidamente serão apresentadas com maior detalhe as fases supracitadas.

4.1.1. Implementação das Classes

Na Figura 28 podemos observar a implementação da classe *AbstractPort*, que é um subelemento da classe *AbstractionDefinition* (ver diagrama de classes apresentado no capítulo 3, Figura 17). Todas as classes implementadas para a representação do *standard* seguem uma estrutura semelhante ao exemplo fornecido. No topo da classe encontram-se todas as variáveis da classe, sejam elas de tipos simples ou compostos, de seguida são implementados um ou mais construtores para a classe, seguem-se os métodos manipuladores e assessores para cada uma das variáveis declaradas e por fim, a função de escrita do nó XML, referente à classe em causa.

Depois de definidas as variáveis e construtores da classe são apresentados os métodos para manipular e aceder às variáveis da classe. Cada um destes métodos está dividido em duas secções, (i) uma responsável por validar o valor que se pretende passar para a variável e (ii) outra para atribuir o valor caso o teste do tipo seja ultrapassado com sucesso. Na Figura 29 podemos visualizar os métodos referentes às variáveis *wirePort_* e *transPort_* da classe *AbstractPort*. No exemplo em questão as variáveis *wirePort_* e *transPort_* são mutuamente exclusivas, ou seja, a classe *AbstractPort* só pode conter um dos dois elementos. Esta restrição advém do elemento de escolha presente no *XML Schema* do IP-XACT. Por este motivo, os manipuladores destas variáveis garantem que se um deles for preenchido com um valor, o outro tem obrigatoriamente que ser nulo, para que o elemento cumpra as restrições do *standard*.

A definição de cada classe termina com a implementação de uma função para a escrita do nó XML correspondente. Nesta função, os dados são novamente submetidos a verificações para garantir que os campos de preenchimento obrigatório contêm dados válidos. Nesta fase assume-se que, se os dados existentes são diferentes do valor por defeito, então são dados válidos, ou seja, as validações executadas nesta função são utilizadas apenas para garantir que os campos obrigatórios estão preenchidos. A função de escrita do nó XML recebe o nó pai e o documento XML a que está associado e no final, a função retorna o nó pai. Desta forma a função escreve todos os nós correspondentes às suas variáveis de tipo simples e associa-as ao nó pai que recebe no momento da sua invocação. Caso as variáveis sejam de tipo composto, é invocada a função de criação do nó desse tipo.

```
class AbstractPort
{
        private string logicalName_;
        private string displayName_;
        private string description_;
        private AbstractWire wirePort_;
        private AbstractTransactional transPort_;
        private VendorExtensions vendorExtensions_;

        //construtores
        public AbstractPort()
        {
                logicalName_ = null;
                displayName_ = null;
                description_ = null;
                wirePort_ = null;
                transPort_ = null;
        }

        public AbstractPort(string logicalName, string
displayName, string description, AbstractWire wirePort)
        {
                logicalName_ = logicalName;
                displayName_ = displayName;
                description_ = description;
                wirePort_ = wirePort;
                transPort_ = null;
        }
}
```

Figura 28 - Excerto da Classe *AbstractPort*

```
//metodos para manipular wirePort

public AbstractWire WirePort
{
        get { return this.wirePort_; }
        set { this.wirePort_ = value; this.transPort_ = null; }
}

//metodos para manipular transPort

public AbstractTransactional TransPort
{
        get { return this.transPort_; }
        set { this.transPort_ = value; this.wirePort_ = null; }
}
```

Figura 29 - Métodos e Assessores com Verificação

```
//criar nodo xml

public XmlNode createNode(XmlDocument doc, XmlNode rootNode)
{
      //rootNode contem o element PORT
      XmlNode node;

      //logicalName node
      if (logicalName_ != null)
      {
           node = doc.CreateNode(XmlNodeType.Element, "spirit",
      "logicalName",
      "http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5");
           node.InnerText = this.logicalName_.ToString().Trim();
           rootNode.AppendChild(node);
      }

      ...

      //wire
      if (wirePort_ != null)
      {
            node = doc.CreateNode(XmlNodeType.Element, "spirit",
"wire", "http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5");
            node = wirePort_.createNode(doc, node);
            rootNode.AppendChild(node);
      }

      return rootNode;
}
```

Figura 30 - Função de Escrita do Nó XML

4.1.2. Implementação da Função de Novo IP

Dependendo do tipo de IP a escrever o elemento base a criar varia. Assim, foi implementada uma função de escrita por cada *top element* do IP-XACT. Na Figura 31 é apresentado um excerto da função de escrita de um IP do tipo *component*.

Nesta função a única verificação realizada é a validação da VLNV do componente. Todas as verificações realizadas aos restantes elementos são executadas no momento de criação dos mesmos. O conjunto de elementos que constituem o VLNV (Vendor, Library, Name e Version) é utilizado pelo *standard* como um identificador único do IP. Para além de ser o id do IP, utilizou-se o VLNV do IP para definir a árvore de diretórios a criar para guardar todos os ficheiros relacionados com o IP. Desta forma torna-se mais fácil a navegação dentro do repositório. Portanto, para garantir a unicidade de cada IP é necessário realizar uma verificação à VLNV do componente a criar. Para além de único, o VLNV, tem de ter todos os campos preenchidos. Assim, a primeira verificação a realizar no momento de criação de um novo IP é a verificação relativa ao preenchimento de todos os campos. Após isso, verifica-se se não existe um ficheiro IP-XACT com a mesma VLNV. Caso exista, o ficheiro não pode ser criado. Caso contrário, o processo de criação do IP pode continuar. Para testar a existência da VLNV é invocada o método *testVLNV*, implementado na classe VLNV. Este método percorre todo o repositório extraindo a VLNV de cada IP, comparando-a depois com a VLNV do IP que se pretende criar. Caso encontre uma VLNV igual, o método retorna *null*, caso não encontre nenhuma VLNV igual retorna o caminho para o diretório onde o ficheiro do IP será criado.

As verificações relacionadas com os elementos presentes no IP são realizadas no momento da sua criação, sendo que no momento da criação do IP a única operação efetuada sobre os elementos é a sua associação ao IP, sem nenhuma verificação adicional. A Figura 32 demonstra um excerto do processo de criação de um subelemento do IP *component*, no caso em questão, um subelemento do tipo *busInterface*. No excerto da Figura 32 é possível identificar duas verificações. Uma que implementa a restrição do IP-XACT que obriga todos os elementos do tipo *busInterface* a ter um nome não nulo, e logo de seguida, uma outra que verifica, caso o nome não seja nulo, se o nome do *busInterface* é único dentro do IP atual. Esta última validação implementa a unicidade do nome dos *busInterfaces* imposta pelo IP-XACT. Se todas as verificações forem ultrapassadas com sucesso o subelemento é criado e adicionado a uma lista que será depois associada ao IP no momento da sua criação.

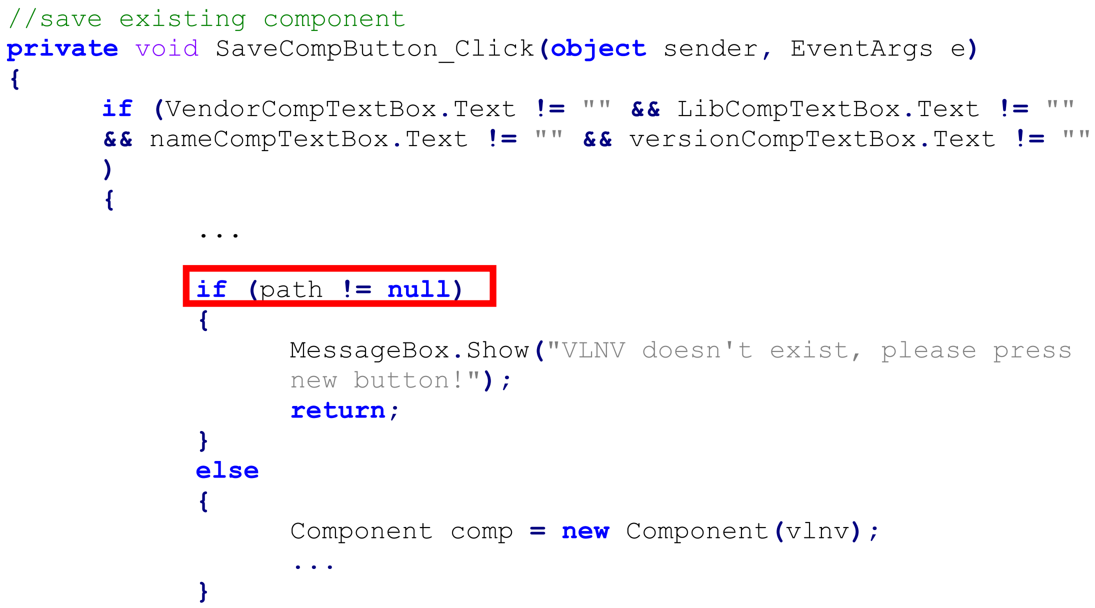
```
private void NewCompButton_Click(object sender, EventArgs e)
{
  if (VendorCompTextBox.Text != "" && LibCompTextBox.Text != ""...)
  {   //criar VLNV
            VLNV vlnv = new
            VLNV(VendorCompTextBox.Text.ToString().Trim(),...,
            "component");
            string path = vlnv.testVLNV();
            if (path == null)
            { MessageBox.Show("VLNV already exists!"); return; }
            else
            {     Component comp = new Component(vlnv);
                  //criar busInterfaces
                  if (busInterfaceList.Count > 0)
                  { foreach (BusInterface bus in busInterfaceList)
                    { comp.busInterface = bus; }}...
```

Figura 31 – Função para Escrever IP

```
//Add busInterface
private void button11_Click(object sender, EventArgs e)
{
      if (busInterfaceNameTextBox.Text != "")
      {
            for (int i = 0; i < busInterfaceList.Count; i++)
            {
                  if (busInterfaceList.ElementAt(i).name ==
            busInterfaceNameTextBox.Text)
                  {
                        DialogResult dialogResult =
                  MessageBox.Show("This interface already
                  exists!\nDoyou want to replace it?", "",
                  MessageBoxButtons.YesNo);
                        if (dialogResult == DialogResult.Yes)
                        {
                              allBusInterComboBox.Items.Remove(busIn
                        terfaceList.ElementAt(i).name);
                              busInterfaceList.Remove(busInterfaceLi
                        st.ElementAt(i));
                        }
                        else
                        { return; } } }
            BusInterface bus = new BusInterface();
            bus.name =
busInterfaceNameTextBox.Text.ToString().Trim();
            ...
            busInterfaceList.AddLast(bus); }
```

Figura 32 - Adicionar Subelemento

No caso de se pretender alterar um IP, o processo é bastante semelhante ao processo de criação. Mais uma vez, a primeira validação a realizar é referente à VLNV do IP. Tal como sucedia no processo de criação de um novo IP, a VLNV tem de estar totalmente preenchida, caso contrário o IP não deve ser alterado. Se esta primeira verificação for ultrapassada devemos verificar se a VLNV existe. Neste caso, se a VLNV existir, a alteração pode ser feita, caso contrário o utilizador deve criar um novo IP. Esta diferença pode ser visualizada comparando a Figura 31 com a Figura 33. Como o processo de identificação dos dados alterados e reescrita apenas das informações alteradas é moroso e complexo, torna-se mais prático reescrever completamente o ficheiro. Por este motivo, quando se guarda alguma alteração num IP reescreve-se por completo o ficheiro, criando-se um ficheiro novo que contém as informações do IP mais as alterações efetuadas.



```
//save existing component
private void SaveCompButton_Click(object sender, EventArgs e)
{
      if (VendorCompTextBox.Text != "" && LibCompTextBox.Text != ""
      && nameCompTextBox.Text != "" && versionCompTextBox.Text != ""
      )
      {
            ...

            if (path != null)
            {
                  MessageBox.Show("VLNV doesn't exist, please press
                  new button!");
                  return;
            }
            else
            {
                  Component comp = new Component(vlnv);
                  ...
            }
```

Figura 33 - Guardar Elemento já existente

4.1.3. Implementação da Função de Leitura de um IP

Na Figura 34 é apresentado um excerto da função de leitura de um IP do tipo *component*, nomeadamente, um excerto referente à leitura do subelemento *channels*. O processo de leitura é efetuado em dois momentos da execução da *framework*. O primeiro momento é o processo de *start up* em que todos os ficheiros no repositório são lidos e depois verificados. O segundo momento é quando é executado um pedido de abertura de um documento IP-XACT. Esta função é bastante simples, limitando-se a percorrer o ficheiro IP-XACT para preencher os vários campos que constituem a estrutura de dados do IP. O processo de leitura não executa verificações. Caso se pretenda determinar se o ficheiro lido é válido é necessário correr a função de validação do respetivo IP, apresentada na secção seguinte.

```
case "spirit:channels":
      Channel channel;
      while (!(reader.NodeType == XmlNodeType.EndElement &&
reader.Name.ToString() == "spirit:channels"))
      {
            reader.Read();
            if (reader.NodeType == XmlNodeType.Element &&
      reader.Name.ToString() == "spirit:channel")
            {
                  channel = new Channel();
                  while (!(reader.NodeType ==
XmlNodeType.EndElement && reader.Name.ToString() ==
"spirit:channel"))
                  {
                        reader.Read();
                        if (reader.NodeType == XmlNodeType.Element)
                         {
                              switch (reader.Name.ToString())
                               {
                                    case "spirit:name":
                                          reader.Read();
                                          channel.name =
                                          reader.Value.ToString();
                                          break;
                                    ...
                              }
                        }
                  }
                  channelsList.AddLast(channel);
            }...
```

Figura 34 - Excerto da Função de Leitura

4.1.4. Implementação da Função de Validação de um IP

A função de validação do IP é utilizada para determinar se o ficheiro IP-XACT lido respeita todas as restrições impostas pelo *standard*. A função de validação só é invocada após um pedido de leitura, para testar o ficheiro que acabamos de ler. Por este motivo existiam duas possibilidades para a sua implementação. Primeiro, podia-se optar por fazer a verificação em paralelo com o processo de leitura, ou seja, à medida que se liam os campos do ficheiro IP-XACT executavam-se as verificações necessárias. Desta forma, quando se chegasse ao fim do ficheiro já se teria um resultado relativamente à integridade do mesmo. No entanto esta opção tornava o processo de leitura muito complexo, obrigando a percorrer o ficheiro mais que uma vez para fazer a procura de referências entre elementos do mesmo ficheiro. Por exemplo, os *physicalPorts* declarados no subelemento *busInterface* têm de corresponder aos portos declarados no elemento *ports* que pertence ao elemento *model*.

Para tornar o processo mais simples optou-se por ler o ficheiro por completo criando-se a estrutura de dados que o representa. Depois disso executa-se uma verificação formal sobre a estrutura de dados para averiguar a sua integridade. Se o ficheiro for inválido é apagado, caso contrário fica disponível para utilização no repositório. Na Figura 35 é apresentado um excerto da função de validação. O excerto apresentado mostra parte das verificações executadas sobre os elementos do tipo *busInterface* presentes no IP do tipo *component*. No código disponibilizado na Figura 35 é possível concluir que, para um *busInterface* ser válido, tem de conter uma referência a um *busDefinition*, representado pelo atributo *busType*, e essa referencia tem de corresponder a um ficheiro IP-XACT presente no repositório. A verificação da existência do IP referenciado é realizada recorrendo à VLNV do IP, através da invocação do método *searchVLNV*, implementada na classe VLNV. É também obrigatória a presença do modo da interface e, caso o *busInterface* em causa contenha um *abstarctType*, os portos físicos associados aos portos lógicos do *abstractType* têm que corresponder a uma referência a um porto declarado no subelemento *ports*, como já foi referido.

A variável *valid* é iniciada com o valor verdadeiro no início da função de validação. Em nenhum outro momento é atribuído a esta variável o valor verdadeiro. Basta que ocorra uma incoerência com o *standard* para que seja atribuído à variável *valid* o valor falso. O processo de validação do IP termina logo após a deteção de uma irregularidade, ou então no fim da estrutura de dados que representa o IP, caso não tenha sido detetado nenhum erro no documento carregado.

Após concluída a validação da estrutura de dados, se o valor presente na variável *valid* for falso, o ficheiro é apagado, juntamente com o diretório, caso este fique vazio.

```
//Validate BusInterfaces
foreach (BusInterface bus in busInterfaceList)
{
      //name
      if (bus.name == null) { valid = false; break; }
      //bus type
      if (bus.busType == null) { valid = false; break; }
      else
      {     bus.busType.setType(GlobalTypes.busDefenition);
            if(!(bus.busType.searchVLNV()))
            {
              valid = false; break; } } ...
      //interface
      if (bus.interfaceMode == null) { valid = false; break; }
      else{ ... }
      //portMaps
      if (bus.abstractionType == null && bus.portCount() > 0)
      { valid = false; }
      for (int i = 0; i < bus.portCount(); i++)
      {     PortMap portMap = bus.getPort(i);
            bool valPort = false;
            foreach (Port port in portList_)
            {
                  if (port.name == portMap.pPort.name)
                  { valPort = true; break; }
            }
            if (!valPort)
            { valid = false; break; } } ...
```

Figura 35 - Excerto da Função de Validação

4.2. Ambiente Gráfico (GUI)

O presente subcapítulo descreve os processos de maior importância envolvidos na implementação do ambiente gráfico dividido em três secções: (i) princípios gerais do ambiente gráfico, que aborda as principais funcionalidades implementadas e explica a implementação dos mecanismos de validação utilizados; (ii) interoperabilidade, onde é apresentada a forma como é realizada a interface com o gestor de repositório e o gerador de código; (iii) por fim a área de desenho, onde são apresentados os mecanismos de deteção de colisão e atualização da área de desenho.

4.2.1. Princípios Gerais

No que diz respeito à criação de *designs* pode optar-se por duas metodologias distintas. Podemos recorrer a uma implementação que siga uma metodologia *bottom-up*, ou optar por uma abordagem *top-down*. Uma abordagem *top-down* implicaria dar suporte à criação de componentes vazios, que seria posteriormente construído à medida das necessidades do projeto, sendo que esses componentes poderiam ser componentes simples (*flat components*), ou componentes hierárquicos (componentes com um *design* associado). Devido às limitações do tempo de desenvolvimento da corrente dissertação optou-se por fornecer suporte ao utilizador para a criação de *designs* seguindo uma abordagem *bottom-up*. A única desvantagem desta abordagem em relação à anterior é obrigar a que todos os componentes que se pretendam utilizar num *design* estejam previamente criados. No entanto, garante que todos os IPs utilizados no *design* são válidos, diminuindo os esforços para verificação de IPs.

O suporte para a abordagem *bottom-up* consiste em mecanismos de criação de barramentos, para definir as interfaces do modelo de topo e mecanismos de instanciação de componentes presentes no repositório. A criação de barramentos foi implementada recorrendo às opções, acessíveis através do clique no botão direito do rato quando este está sobre a área de desenho, e através do menu *tools->New Bus*, na parte superior da janela da *framework*. A instanciação de componentes foi implementada recorrendo a um mecanismo de *drag-and-drop*. O utilizador clica com o botão esquerdo do rato sobre um dos elementos disponíveis na lista de IPs do repositório mantendo o botão premido, selecionando assim o elemento. Se no momento em que se solta o botão do rato, este se encontrar sobre a área de desenho, o elemento selecionado na lista de IPs é carregado para o *design* e desenhado na posição atual do rato.

Independentemente da abordagem de construção dos *designs* existe um conjunto de verificações que é necessário executar para evitar ao máximo que o utilizador desenvolva sistemas com erros. A primeira verificação é realizada no momento em que o programa é arrancado e consiste na validação de todos os IPs do repositório. Isto permite que o repositório se mantenha intacto e que os elementos presentes na lista de IPs disponíveis para instanciação nos *designs* se encontrem devidamente validados. Uma outra verificação importante está relacionada com as conexões entre os diversos IPs que constituem o *design*. Optou-se por implementar esta verificação em *design time*, ou seja, sempre que o utilizador tenta estabelecer uma conexão, uma validação é executada, sendo que, só se este teste for ultrapassado a conexão é executada. Desta forma o utilizador é impossibilitado de realizar uma conexão entre duas interfaces diferentes, impedindo-o de introduzir erros no *design*. A Figura 36 representa um excerto do código responsável por realizar a validação supracitada.

```
switch (bus.Bus.interfaceMode.mode)
{
  case "master":
    if (compBus.Bus.busType.VLNVtoString() !=
bus.Bus.busType.VLNVtoString() || !(compBus.Bus.interfaceMode.mode ==
"mirroredMaster" || compBus.Bus.interfaceMode.mode == "slave")){
      MessageBox.Show("Error");
      return;
    }
    break;
...
  case "slave":
    if (compBus.Bus.busType.VLNVtoString() !=
bus.Bus.busType.VLNVtoString() || !(compBus.Bus.interfaceMode.mode ==
"mirroredSlave" || compBus.Bus.interfaceMode.mode == "master")){
      MessageBox.Show("Error");
      return;
    }
  break;
...
}
```

Figura 36 - Excerto do código de validação de conexão

Como se pode visualizar, o processo de validação tem em conta dois campos do *standard* IP-XACT. Primeiro, para que uma conexão entre duas interfaces seja válida, estas devem partilhar o mesmo tipo de barramento (campo *busType*). Para além do tipo de barramento é ainda necessário que as interfaces sejam opostas, ou seja, caso a primeira interface envolvida na conexão seja *master*, a segunda interface tem obrigatoriamente que ter uma interface *mirrorred master* ou *slave*, caso contrário a conexão não cumpre os requisitos e não será efetuada.

Por fim, optou-se por implementar um mecanismo de seleção de repositórios. Este mecanismo permite a existência de vários repositórios, no entanto apenas um pode ser utilizado de cada vez. O programa é iniciado por defeito com o último repositório utilizado, sendo que na primeira vez, é apresentado uma caixa de diálogo para seleção do repositório. O objetivo desta funcionalidade é aumentar o grau de organização dentro de cada repositório, dando a possibilidade de criação de um repositório por projeto ou por utilizador.

No capítulo 3.1.2 referiu-se que este módulo seria o responsável por interligar o restante sistema. O subcapítulo seguinte clarifica a implementação realizada, explicando os mecanismos utilizados para conectar o módulo de gestão do repositório e o módulo dos geradores ao ambiente gráfico.

4.2.2. Interoperabilidade Entre Módulos

A estrutura da presente dissertação tentou seguir uma metodologia o mais modular possível, para que fosse possível a implementação de diversos módulos independentes entre si. Esta metodologia permite a criação e teste de sistemas de forma independente, reduzindo a complexidade do projeto e facilitando os processos de deteção de erros. Este pensamento levou a que, no capítulo Capítulo 3, a *framework* fosse dividida em três módulos: (i) gestor de repositório; (ii) interface gráfico; (iii) geradores. Apesar de independentes, é necessário que um destes módulos implemente uma interface, para que se obtenha o sistema final. Por ser responsável pela interface com o utilizador, o módulo da interface gráfica é o mais indicado para efetuar a interface entre os restantes módulos. Utilizando a interface gráfica como ponto de ligação consegue-se, de uma forma simples e intuitiva, dar acesso ao utilizador aos restantes módulos, de uma forma transparente ou através de um clique num botão.

Na Figura 37 é possível visualizar a interface entre o ambiente gráfico e o gestor de repositório. Esta interface consiste em disponibilizar ao utilizador uma lista com todos os IPs disponíveis para instanciação. Em primeiro lugar é criado um nó, que será a raiz da árvore de diretórios, e é-lhe passado o valor da raiz do repositório. De seguida a função *createTreeView* é invocada com o nó da árvore e a informação relativa à raiz do repositório como argumentos. Esta função é recursiva e tem como objetivo percorrer todos os diretórios dentro do repositório para construir uma lista com todos os ficheiros IP-XACT disponíveis, acrescentando-os ao nó raiz. Após o retorno da função, em *root* temos uma lista de todos os ficheiros IP-XACT do repositório, organizados por diretórios. Para concluir o processo, acrescenta-se este nó ao conjunto de nós que constituem o objeto *repositoryView*.

```
DirectoryInfo dirInfo = new DirectoryInfo(GlobalTypes.rootDir);
TreeNode root = new TreeNode(dirInfo.FullName);
createTreeView(root, dirInfo);
repositoryView.Nodes.Add(root);
```

Figura 37 - Construção da árvore com os componentes IP-XACT do repositório

Para além desta interface, que é visível para o utilizador, o diagrama de classes e os métodos implementados no subcapítulo 4.1, dão suporte a esta interface de forma transparente, possibilitando que o ambiente gráfico carregue, crie e altere documentos IP-XACT presentes no repositório. Todo o processo de gestão do repositório encontra-se dividido pelas várias classes que representam o *standard*, pelo que a utilização destas classes por parte do ambiente gráfico oferece acesso direto a todas as funcionalidades do gestor de repositório. Sempre que um IP é instanciado é executada uma operação de leitura do ficheiro IP-XACT desse IP, para recolher todas as informações relevantes como o número, nome e tipo de barramentos que constituem o IP e só depois se procede ao desenho da sua representação gráfica. Este processo de comunicação com o repositório sempre que um IP é instanciado é crucial para que os processos de verificação possam ser executados.

A Figura 38 ilustra a interface entre o ambiente gráfico e o gerador de código. Esta interface é realizada recorrendo a uma nova janela que recebe como argumentos uma lista com todos os ficheiros HDL dos IPs instanciados no *design* atual, as informações relacionadas com o ficheiro do IP que se está a desenhar, o caminho para o documento IP-XACT do IP e o nome dado ao *design*. Estas informações são depois utilizadas para gerar o código HDL do IP desenhado na área de desenho do interface gráfico. Os detalhes relacionados com a implementação dos geradores estão presentes no subcapítulo 4.3, no entanto é de salientar o facto dos argumentos recebidos pela interface com os geradores. Todos os argumentos são *string*, à exceção do *fileInfo*. No entanto este é um tipo interno do C#, que pode ser substituído por um conjunto de *strings* que caracterizem o ficheiro e o diretório a que este pertence. Esta interface, independente das classes que constituem a arquitetura da *framework*, torna os geradores implementados independentes da aplicação em que são utilizados.

```
FileInfo fileInfo = new FileInfo(designPath);
DesignGeneratorForm designGen = new DesignGeneratorForm(
verilogFiles, fileInfo, componentPath, NameTextBox.Text);
designGen.ShowDialog();
```

Figura 38 - Interface entre o *Design* e o gerador de código

A implementação do ambiente gráfico termina com a descrição dos principais mecanismos implementados na área de desenho. O subcapítulo seguinte aborda os mecanismos de atualização da área de desenho, deteção de sobreposições entre IPs e desenho das conexões efetuadas, por serem considerados como os mecanismos mais relevantes.

4.2.3. Área de Desenho

Por forma a tornar a experiencia de utilização a mais prática possível para o utilizador, existe um conjunto de mecanismos que devem ser implementados dentro da área onde o desenho do sistema é efetuado. Alguns desses mecanismos são o *highlight* dos elementos selecionados e a realocação de elementos caso exista alguma sobreposição. Os mecanismos de *highlight* devem ser implementados na classe que representa o objeto. No caso de um IP este mecanismo é implementado na classe *ObjComponent*, mas no caso das conexões o *highlight* é implementado no evento de clique sobre a área de desenho.

A realocação de IPs devido a sobreposição é implementada na área de desenho e consiste em determinar se dois ou mais IPs partilham uma ou mais coordenadas dentro da tela. Analisando o exemplo dado pela Figura 39 é possível obter a condição para a determinação da sobreposição entre dois retângulos. Sejam R1 e R2 a representação de dois IPs instanciados na área de desenho. V1 e V2 são dois vértices de R1 e V3 e V4 dois vértices de R2. Todos os vértices são descritos por um par de coordenadas (x,y). Os dois retângulos não estão sobrepostos se R1 está à direita de R2 ($V1.x > V4.x$), à esquerda ($V2.x < V3.x$), acima ($V2.y < V3.y$), ou abaixo ($V1.y > V4.y$). Basta que uma destas condições se verifique para que a possibilidade de sobreposição seja eliminada. Por exemplo se $V1.x > V4.x$, mesmo que os retângulos tenham coordenadas em Y coincidentes, não existirá sobreposição. A mesma análise pode ser efetuada para os restantes casos. Assim, no final da análise do exemplo apresentado é possível chegar à condição apresentada em **(19)**, que garante a inexistência de sobreposição entre dois elementos.

$$\boldsymbol{(V1.x>V4.x \;||\; V2.x<V3.x \;||\; V1.y>V4.y \;||\; V2.y<V3.y)} \tag{19}$$

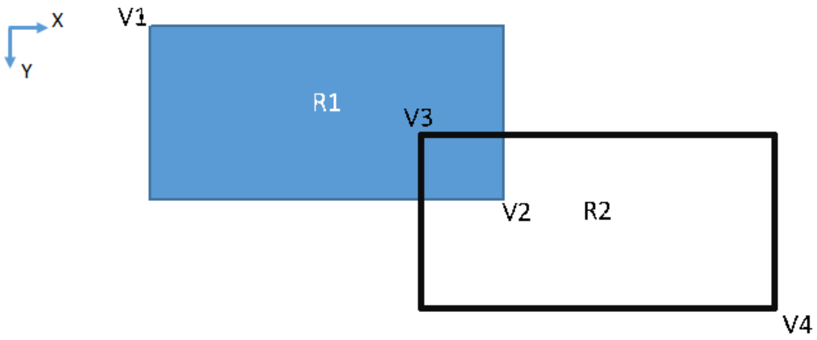


Figura 39 - Exemplo para deteção de colisão

O C# oferece um método, pertencente à classe *Rectangle* com o nome *IntersectsWith* cuja funcionalidade é semelhante ao exemplo acima citado. Por uma questão prática optou-se pela utilização dos métodos já oferecidos pelos objetos C#. Assim, a Figura 40 apresenta o excerto de código implementado para determinar a colisão entre dois IPs. Cada IP é representado por uma *pictureBox*. Através do método *ClientRectangle* é possível obter o retângulo que representa a *pictureBox* de cada IP e a partir daí usar o método *ClientRectangle*. De notar que o método retorna o retângulo de cada IP com coordenadas na origem, por isso é necessário somar a essas coordenadas um *offset* que corresponde à posição do IP na tela de desenho através do método *Offset*. Caso seja detetada uma colisão, o IP que foi movido é sujeito a uma realocação e a função de deteção de colisão é novamente invocada para determinar se a realocação não originou outro caso de sobreposição. O processo repete-se até que seja encontrada uma posição onde não existe sobreposição.

```
foreach (ObjComponent comp in design_.Components){
  if (comp != component){
    Rectangle compRect2 = comp.ComponentImage.ClientRectangle;
    compRect2.Offset(comp.ComponentImage.Location);
    if (compRect1.IntersectsWith(compRect2)){
      component.setLocation(compRect2.Location.X,
compRect2.Location.Y + 10 + compRect2.Height);
      DesignSpace.DetectColision(component, designArea);
    }
  }
}
```

Figura 40 - Função para deteção de colisão

O processo de desenho de uma conexão é bastante diferente do processo de instanciação de um IP. Enquanto a instanciação de um IP implica a criação de um novo objeto gráfico, neste caso uma *pictureBox*, no caso de uma conexão apenas é desenhada uma linha sobre a tela de desenho para conectar os dois extremos dos objetos gráficos. A informação relacionada com a conexão é guardada numa classe com o nome *ObjConnection*, que contem a informação relativa ao ponto de origem e de destino da conexão. Esta classe é também responsável por calcular automaticamente os vértices da conexão, através das coordenadas dos pontos de origem e destino e do lado, direito ou esquerdo, em que estes pontos se encontram. Portanto, como não existe nenhum elemento gráfico associado à conexão é necessário atualizar o desenho presente na tela sempre que uma operação sobre um elemento é efetuado. Por exemplo, para apagar uma conexão é efetuada uma operação de desenho de conexão com a cor do traço igual à cor do fundo da área de desenho e o objeto com a informação da conexão é eliminado. Estas particularidades associadas às conexões implicam que, quando existem situações em que se apaga uma conexão que intersecta uma outra, a segunda conexão ficará mal desenhada porque o ponto de interceção é apagado. Duas possíveis situações visando a resolução deste problema envolvem: (i) algoritmo para determinação e redesenho dos pontos de interceção envolvidos; (ii) redesenhar todas as ligações que não foram apagadas sem calcular pontos de interceção. Um estudo das abordagens supracitadas é apresentado de seguida e na Figura 43 é possível visualizar um excerto do código da abordagem selecionada.

Em relação à primeira possibilidade, o algoritmo para a determinação e redesenho dos pontos de interceção, analisemos a Figura 41, que apresenta os casos possíveis de sobreposição.

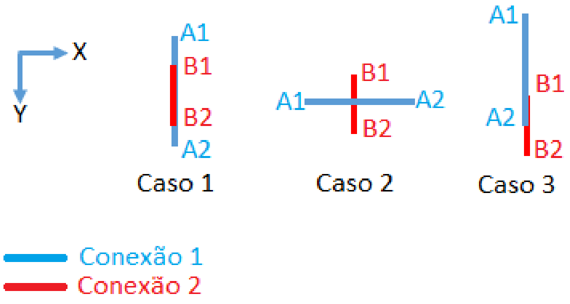


Figura 41 - Casos de colisão entre conexões

A partir dos cenários apresentados é possível chegar a um conjunto de condições que permitem determinar os casos em que existe sobreposição. Para todos os casos existem duas situações: (i) a coordenada X é constante e Y varia do ponto de origem para o ponto de destino; ou (ii), a coordenada Y é constante e X varia do ponto de origem para o ponto de destino. Para os casos 1 e 3, com as retas na vertical tal como se encontram representadas na Figura 41, existe uma colisão caso a condição **(20)** se verifique. Na situação em que as retas se encontram na horizontal, a condição para a existência de colisão encontra-se descrita em

(21). Em ambos os casos apresentados existe sobreposição de mais que um ponto. O intervalo de sobreposição é dado pelos dois pontos que não são o máximo nem o mínimo do conjunto constituído pelos quatro pontos que constroem as duas retas a avaliar. Na situação ilustrada no caso 2 apenas um ponto está envolvido na sobreposição das duas retas. A condição para a determinação da interceção das duas retas é dada por **(22)**. Para este tipo de situações a coordenada do ponto de interceção é dada pelos valores de x e y que se mantêm constantes, ou seja, o ponto de interceção é dado pelo par de coordenadas (A1.y, B1.x).

$$!(A2.y < B1.y \;\|\; A1.y > B2.y) \;\&\&\; (A1.x == B1.x) \tag{20}$$

$$!(A2.x < B1.x \;\|\; A1.x > B2.x) \;\&\&\; (A1.y == B1.y) \tag{21}$$

$$\begin{gathered} (A1.x < B1.x \;\&\&\; A2.x > B1.x) \\ \&\& \\ (A1.y < B2.y \;\&\&\; A1.y > B1.y) \end{gathered} \tag{22}$$

A segunda possibilidade passa por eliminar a conexão que se pretende apagar da lista de conexões e de seguida redesenhar todas as conexões na lista. A conexão selecionada encontre-se guardada numa variável com o nome *connectionSelected_*. Se a tecla de *delete* for pressionada e esta variável for diferente de *null* significa que existe uma conexão selecionada que é necessário apagar. Para isso, em primeiro lugar remove-se a conexão da lista de conexões e de seguida desenha-se uma conexão igual à que se pretende apagar com a cor de fundo da área de desenho. Estas operações são comuns para os dois cenários apresentados. A diferença está nas operações seguintes. Na primeira possibilidade era necessário realizar um conjunto de condições para determinar a existência de sobreposição e de seguida era necessário determinar os pontos de sobreposição. Isto obrigava a um ciclo para percorrer a lista de conexões para posterior comparação de todas as conexões com a conexão que foi apagada. No segundo caso também é necessário um ciclo para percorrer todas as conexões existentes, mas em vez de se efetuar comparações com a conexão apagada, redesenha-se todos os pontos das conexões que se encontram na lista.

Optou-se pela implementação da segunda possibilidade por oferecer um desempenho superior e uma menor complexidade de código. O resultado da implementação pode ser visualizado no excerto apresentado na Figura 42.

```
if (connectionSelected_ != null)
{
   design_.Connections.Remove(connectionSelected_);
   connectionSelected_.setConnectionColor(designArea.BackColor);
   connectionSelected_.DrawLine(designArea);
   connectionSelected_.setConnectionColor(Color.Black);

   //remove the connection in the datagridview to sync the
   // design area and the datagrid
   ...
   connectionSelected_ = null;
   foreach (ObjConnection connection in design_.Connections)
   {
      connection.DrawLine(designArea);
   }
}
```

Figura 42 - Excerto de código para redesenho de conexões

No que à atualização da área de desenho diz respeito, optou-se pela utilização dos eventos disponibilizados pelo C#. Estes eventos são desencadeados automaticamente sempre que ocorre uma operação que exige que a área de desenho seja atualizada, como por exemplo, uma ação de *scroll* ou de realocação de um IP. Durante este processo existe a necessidade de: (i) apagar o desenho de todas as conexões, uma vez que os pontos de origem e destino podem ter sido alterados; (ii) atualizar a posição de todos os IPs instanciados; (iii) redesenhar todas as conexões com as novas coordenadas já atualizadas. A Figura 43 ilustra os passos supracitados através de um excerto do código implementado para atualização da área de desenho.

```
//Erase all connections
foreach (ObjConnection connection in design_.Connections){
  connection.setConnectionColor(designArea.BackColor);
  connection.DrawLine(designArea);
  connection.setConnectionColor(Color.Black);
}
//Refresh each connection origin and destination positions
foreach (ObjComponent component in design_.Components){
  foreach (ObjBus bus in component.Buses){
    foreach (ObjConnection connection in design_.Connections){
      if (component.InstanceName == connection.Comp1 && bus.BusName ==
connection.Bus1){
        connection.setOrigin(
bus.Parent.ComponentImage.Location.X + bus.BusImage.Location.X,
bus.Parent.ComponentImage.Location.Y + bus.BusImage.Location.Y + 10,
bus.Side);}...
    }}}...
//Re-draw all connections
foreach (ObjConnection connection in design_.Connections){
  connection.DrawLine(designArea);
}
```

Figura 43 - Excerto da função de atualização da área de desenho

4.3. Gerador de Código

A geração de código assenta na implementação de *stylesheets* XSLT para transformação dos ficheiros XML IP-XACT em ficheiros de código Verilog e ficheiros de restrições de implementação. Enquanto o código Verilog é igual, independentemente da plataforma alvo, os ficheiros responsáveis por definir as restrições do projeto, no caso da Xilinx os ficheiros UCF e no caso da Altera os ficheiros TCL, apresentam diferenças significativas, o que obriga à implementação de um *stylesheet* diferente para cada distribuidor.

Nas secções seguintes serão abordadas as implementações dos *stylesheets* comuns a ambos os distribuidores, seguidos das implementações da geração automática dos ficheiros responsáveis pelas restrições dos projetos e do método de invocação dos *stylesheets* por parte da *framework.*

4.3.1. XSLT para Geração de Ficheiros HDL

O primeiro aspeto a considerar na implementação dos XSLT responsáveis pela conversão do *standard* IP-XACT para código HDL é a nomenclatura da linguagem HDL que será gerada. Como referido no anterior capítulo a linguagem HDL a que a *framework* dá suporte é o Verilog. Nesse sentido, cada documento inicia-se com a definição da escala de tempo seguido de possíveis parâmetros e constantes para o módulo em causa. O módulo inicia-se depois através da palavra reservada *module* seguida do nome que se pretende dar ao mesmo. Os sinais de *input* e *output* do sistema são depois referidos, podendo a sua instanciação ser realizada dentro ou fora dos parenteses do módulo. Na implementação apresentada optou-se por indicar apenas o nome dos portos de I/O dentro dos parenteses do módulo, sendo a sua instanciação realizada dentro do corpo do módulo. Na Figura 44 é possível visualizar o excerto de código XSLT responsável pela instanciação de um módulo. A função *position()* retorna o índice do nó que está a ser testado atualmente enquanto a função *last()* dá o índice do último nó do conjunto que está a ser testado. Neste caso o conjunto definido pela expressão XPath é constituído por todos os nós do tipo *spirit:port* presentes em *spirit:ports*.

```
<xsl:for-each select="$baseCompDoc/spirit:component">
   module <xsl:value-of select="spirit:name"/>(
   <xsl:for-each select="spirit:model/spirit:ports/spirit:port">
      <xsl:value-of select="spirit:name"/>
      <xsl:if test="position() != last()">,</xsl:if>
   </xsl:for-each>
);
```

Figura 44 - XSLT para instanciação de módulo

No caso em que o objeto IP-XACT a transformar é do tipo componente não é necessário realizar grandes operações sobre o ficheiro XML. No entanto, quando o objeto a transformar é do tipo *design* é necessário um conjunto de operações auxiliares até que se obtenha os ficheiros necessários para a criação do correspondente Verilog.

O primeiro passo consiste em correr o ficheiro XML de *design* para identificar todos os componentes que de forma direta ou indireta se encontram conectados. Como já foi referido no capítulo 3 secção 3.1.3.2, as ligações mapeadas no ficheiro XML não têm uma ordem predefinida mas o XSLT a aplicar tem de ser capaz de criar sempre o mesmo ficheiro, independentemente da ordem com que as ligações aparecem dispostas. Para além do ficheiro de *design* são necessários os ficheiros XML dos componentes instanciados no *design*, assim como o ficheiro XML que representa o componente hierárquico que o *design* define. Como o XSLT é para ser aplicado a um ficheiro XML específico a única forma de ter acesso aos restantes XMLs necessários é através do seu carregamento dinâmico no momento da transformação. Isso é conseguido através da utilização do elemento *variable* juntamente com a função *document()* (ver Figura 45). Graças a este método consegue-se obter uma variável cujo valor é a raiz do documento XML carregado através da função *document()*. Depois disso, é possível aplicar qualquer tipo de regra sobre a variável, como se de um documento XML se tratasse.

```
<xsl:variable name="compPath">
    <xsl:value-of select="$rootPath"/><xsl:value-of
select="./path"/>
</xsl:variable>
<xsl:variable name="compDoc" select="document($compPath)"/>
```

Figura 45 - Carregamento dinâmico de documento XML

A necessidade de executar o carregamento dinâmico de ficheiros XML obriga a que os caminhos para os ficheiros sejam construídos com base num algoritmo conhecido. Desta forma, a utilização das *stylesheets* obriga a que se estabeleçam as seguintes regras:

1. Os ficheiros dos componentes IP-XACT presentes na instanciação de um *design* devem encontrar-se guardados num diretório cujo caminho é dado por *raiz_do_repositório\vendor\library\name\version*;
2. O nome do ficheiro IP-XACT do componente deve ser construído através do nome e versão do mesmo, seguindo a estrutura *name.version.xml*;
3. Como o ficheiro sobre o qual se invoca o XSLT é o ficheiro componente do componente hierárquico do *design*, o nome dado ao ficheiro de *design* deve também seguir um modelo. Assim, o ficheiro IP-XACT de *design* deve estar guardado no mesmo diretório do respetivo ficheiro componente IP-XACT e o seu nome deve seguir o modelo *name.design.version.xml*.

No capítulo 3 foram introduzidas os elementos *function* e *sequence*, disponíveis apenas na versão 2.0 do XSLT como a solução para percorrer o ficheiro de *design* de forma recursiva e assim determinar todas as ligações diretas e indiretas entre componentes. No entanto o Visual Studio não oferece suporte à utilização da versão 2.0 do XSLT. As bibliotecas apenas suportam o XSLT 1.0. Para resolver o problema do suporte é necessário instalar uma API capaz de processar *stylesheets* na versão 2.0. Na implementação do gerador de código da corrente dissertação optou-se pela utilização da API da Saxon, atualmente na versão 9.5 disponibilizada desde Abril de 2013 [44].

O pacote disponibilizado pela Saxon é uma coleção de ferramentas para o processamento de ficheiros XML que inclui [45]: um processador para XSLT 2.0 e 1.0 que pode ser utilizado pela linha de comandos ou então invocado por uma aplicação através das suas APIs; um processador para XPath 2.0 acessível para aplicações através da API; um processador para XQuery 1.0 passível de ser utilizado pela linha de comandos ou através da API.

Para realizar a integração da Saxon no Visual Studio é necessário acrescentar aos recursos do projeto a API da Saxon. Depois de incluída no projeto basta adicionar a biblioteca da Saxon através do comando *using Saxon.API*. A utilização da API da Saxon torna o processo de transformação mais lento, relativamente ao processador XSLT suportado pelo Visual Studio.

Por este motivo as *stylesheets* XSLT 2.0 são utilizadas apenas no processo de geração dos ficheiros intermédios de *design*, sendo a transformação do ficheiro intermédio de *design* em código HDL realizada recorrendo novamente ao XSLT 1.0.

A Figura 46 representa o código da função recursiva utilizada para percorrer o ficheiro XML de *design* de forma a identificar os componentes ligados entre si. A função é invocada com o parâmetro raiz, que representa o elemento *spirit:interconnections*, com o parâmetro *start*, que indica nó *spirit:interconnection* que se está a analisar e com o parâmetro *list*, que contêm a lista de todos os elementos já encontrados. Quando invocada, a função guarda na variável *connection* todos os nós que respeitam a condição definida no atributo *select*. Por fim, caso existam elementos na lista *connection* que ainda não estejam na lista *list*, a função é novamente invocada com a raiz, o nó atual de *connection* e a lista *list* mais o nó atual. A função é invocada tantas vezes como o número de nós existentes em *connection* que não estejam presentes em *list*. O elemento *sequence* permite que a função retorne o resultado obtido, guardando os componentes interligados numa lista.

```
<xsl:function name="esrg:find-connections">

   <xsl:param name="root"/>
   <xsl:param name="start"/>
   <xsl:param name="list"/>

   <xsl:variable name="connection"
select="$root/spirit:interconnection/spirit:activeInterface[(preceding
-sibling::spirit:activeInterface/@spirit:componentRef =
$start/@spirit:componentRef and preceding-
sibling::spirit:activeInterface/@spirit:busRef =
$start/@spirit:busRef) or (following-
sibling::spirit:activeInterface/@spirit:componentRef =
$start/@spirit:componentRef and following-
sibling::spirit:activeInterface/@spirit:busRef =
$start/@spirit:busRef)]"/>

   <xsl:sequence select="$start | ($connection except
$list)/esrg:find-connections($root, ., . | $list)"/>

</xsl:function>
```

Figura 46 - Função recursiva para determinação de ligações

Devido à necessidade de execução da a função acima apresentada por cada *spirit:activeInterface*, o ficheiro auxiliar final apresenta algumas repetições, desnecessárias para o processo de geração do código HDL. Assim um outro ficheiro auxiliar foi criado graças a outra *stylesheet* cuja única funcionalidade é reproduzir o ficheiro gerado anteriormente sem repetições. O ficheiro resultante desta operação é depois utilizado para a geração do ficheiro HDL do componente hierárquico.

A criação de um ficheiro Verilog de um componente hierárquico envolve a instanciação do modelo hierárquico superior, tal como acontece para um componente simples, a crição de variáveis *wire* para realizar as conexões entre os submódulos instanciados, a instanciação dos submódulos que constituem o componente e mapeamento dos seus sinais dentro do componente. O processo de instanciação do modelo hierárquico superior é em tudo semelhante à instanciação de um módulo de um componente simples (ver Figura 44). A instanciação dos submódulos é realizada através da iteração ao longo dos elementos componentInstance presentes no documento auxiliar de *design*. Por cada nó *componentInstance* carrega-se o documento XML do componente para retirar do campo *spirit:name* o tipo do módulo a instanciar. O nome do submódulo é dado pelo campo *instanceName*. As ligações entre os portos dos submódulos e os *wire* definidos no modelo hierárquico superior são realizadas uma a uma no momento de instanciação dos portos dos submódulos. Este processo é ilustrado pela Figura 47.

```
<xsl:for-each select="./design/componentInstances/componentInstance">
   <xsl:variable name="instanceName" select="./name"/>
   <xsl:variable name="compPath">D:\Thesis\
      <xsl:value-of select="./path"/>
   </xsl:variable>
   <xsl:variable name="compDoc" select="document($compPath)"/>
   <xsl:value-of select="$compDoc/spirit:component/spirit:name"/>
   <xsl:text> </xsl:text> <xsl:value-of select="$instanceName"/>(
   <xsl:for-each
select="$compDoc/spirit:component/spirit:busInterfaces/spirit:busInter
face">
     <xsl:variable name="bus" select="./spirit:name"/>
     <xsl:variable name="busIndex" select="position()"/>
     <xsl:variable name="lastBusIndex" select="last()"/>
     <xsl:for-each select="./spirit:portMaps/spirit:portMap">
       .<xsl:value-of select="./spirit:physicalPort/spirit:name"/>(
        <xsl:call-template name="find-port">
         <xsl:with-param name="index" select="position()"/>
         <xsl:with-param name="baseComp" select="$baseCompDoc"/>
         <xsl:with-param name="busRef" select="$bus"/>
         <xsl:with-param name="instanceName" select="$instanceName"/>
            <xsl:with-param name="designDoc" select="$designDoc"/>
          </xsl:call-template>)
          <xsl:if test="$busIndex != $lastBusIndex">,</xsl:if>
          <xsl:if test="$busIndex = $lastBusIndex">
            <xsl:if test="position() != last()">,</xsl:if></xsl:if>
       </xsl:for-each>
   </xsl:for-each>);
</xsl:for-each>
```

Figura 47 - Instanciação dos submódulos

O processo de determinação da conexão a estabelecer depende do tipo de ligação definido no documento de *design*. Caso a ligação seja do tipo spirit:interconnection, ligação entre dois submódulos dentro do modelo hierárquico superior, é necessário construir o nome do *wire* da mesma forma como ele foi construído no momento da sua declaraçaõ (Figura 48 parte superior). Caso a ligação seja entre um sinal do modelo hierárquico superior e um porto de um submódulo, ligação do tipo *spirit:hierConnection*, o nome é dado pelo nome do sinal do modelo hierárquico superior (Figura 48 parte inferior).

```
<xsl:for-each
select="$doc/spirit:component/spirit:busInterfaces/spirit:busInterface
">
   <xsl:if test="./spirit:name=$busRef">
   <xsl:for-each select="./spirit:portMaps/spirit:portMap">
     <xsl:if test="position() = $index">
       <xsl:value-of select="$component"/>_
       <xsl:value-of select="$busRef"/>_
       <xsl:value-of select="./spirit:physicalPort/spirit:name"/>
     </xsl:if>
   </xsl:for-each>
   </xsl:if>
</xsl:for-each>


<xsl:for-each
select="$designDoc/design/hierConnections/hierConnection">
   <xsl:variable name="topBus" select="./busRef"/>
   <xsl:if test="./interface/@busRef = $busRef and
./interface/@componentRef=$instanceName">
     <xsl:for-each
select="$baseComp/spirit:component/spirit:busInterfaces/spirit:busInte
rface">
       <xsl:if test="$topBus = spirit:name">
       <xsl:for-each select="spirit:portMaps/spirit:portMap">
         <xsl:if test="position() = $index">
          <xsl:value-of select="spirit:physicalPort/spirit:name"/>
         </xsl:if>
       </xsl:for-each>
       </xsl:if>
     </xsl:for-each>
   </xsl:if>
</xsl:for-each>
```

Figura 48 - Determinação do nome da ligação

Uma vez que o *standard* IP-XACT não define comportamento, apenas interfaces, as transformações geradas automaticamente contêm apenas instanciações e mapeamentos entre módulos. Por esse motivo, após a geração dos ficheiros HDL, o utilizador deve editar os mesmos, a fim de acrescentar o comportamento desejado ao componente.

Explicados os mecanismos para a geração dos ficheiros Verilog é apresentado, nas secções seguintes, o processo de geração dos ficheiros com as restrições do projeto de *hardware*, para a Xilinx e para a Altera, pela respetiva ordem.

4.3.2. XSLT para Geração de Ficheiros UCF

Para além dos ficheiros HDL, a geração de um *bitstream* para programação de uma FPGA requer a existência de ficheiros que implementem as restrições do projeto, como as restrições de tempo e o mapeamento dos sinais aos respetivos portos da FPGA. No caso da Xilinx, esses ficheiros são do tipo UCF. Nesse sentido, no momento em que a ordem de geração de código é dada, após a aplicação do primeiro *template* para transformação do ficheiro XML em código Verilog, é aplicado um segundo *template*, de novo sobre o ficheiro do componente IP-XACT, com o objetivo de criar o ficheiro UCF correspondente. Este ficheiro contém todos os portos do componente, prontos para serem mapeados. Desta forma, o utilizador apenas tem de completar o mapeamento do sinal com o pino da FPGA pretendido. A Figura 49 é parte do ficheiro XSLT responsável pela geração automática do ficheiro UCF. Como podemos visualizar na imagem, por cada porto presente no componente é escrita a linha *NET nome_do_porto LOC= ;*. Caso o porto seja constituído por mais que um bit, é invocado o *template* com o nome *vector*, responsável por gerar a NET para cada um dos bits do porto.

```
<xsl:template match="/">
      <xsl:for-each
      select="spirit:component/spirit:model/spirit:ports/spirit:port">
            <xsl:if test="not(spirit:wire/spirit:vector)">
            NET "<xsl:value-of select="spirit:name"/>" LOC = ;
            </xsl:if>
            <xsl:if test="spirit:wire/spirit:vector">
            <xsl:call-template name="vector">
                  <xsl:with-param name="name" select="spirit:name"/>
                  <xsl:with-param name="lBits"
                  select="spirit:wire/spirit:vector/spirit:left"/>
                  <xsl:with-param name="rBits"
                  select="spirit:wire/spirit:vector/spirit:right"/>
            </xsl:call-template>
            </xsl:if>
      </xsl:for-each>
</xsl:template>
```

Figura 49 - Excerto do XSLT para geração de ficheiro UCF

4.3.3. XSLT para Geração de Ficheiros TCL

No caso da Altera, são os ficheiros do tipo TCL os responsáveis pela implementação das restrições de um projeto de *hardware*. Estes ficheiros definem não só o mapeamento entre os sinais e os portos das placas, como também a família e dispositivo alvo do projeto a criar, os ficheiros HDL envolvidos no projeto e restrições de tempo e de colocação dos diversos blocos lógicos no momento das operações de *placement & routing*. A *stylesheet* para a criação de ficheiros TCL é bastante semelhante à *stylesheet* supracitada para o caso da Xilinx. Como se pode observar na Figura 50, a diferença reside na existência de um conjunto de restrições globais relacionadas com a família e dispositivos alvo do projeto e na linha que é necessário escrever para mapear um porto. No caso da Altera, a linha a escrever tem o formato *set_location_assignment -to nome_do_porto*.

```
<xsl:template match="/">
    project_new <xsl:value-of select="$projName"/> -overwrite

    set_global_assignment -name FAMILY <xsl:value-of
select="$family"/>
    set_global_assignment -name DEVICE <xsl:value-of
select="$device"/>
    set_global_assignment -name VERILOG_FILE <xsl:value-of
select="$verilogFileName"/>

    <xsl:for-each
select="spirit:component/spirit:model/spirit:ports/spirit:port">
      <xsl:if test="not(spirit:wire/spirit:vector)">
    set_location_assignment -to <xsl:value-of
select="spirit:name"/><xsl:text>
</xsl:text>
      </xsl:if>
      <xsl:if test="spirit:wire/spirit:vector">
        <xsl:call-template name="vector">
          <xsl:with-param name="name" select="spirit:name"/>
          <xsl:with-param name="lBits"
select="spirit:wire/spirit:vector/spirit:left"/>
          <xsl:with-param name="rBits"
select="spirit:wire/spirit:vector/spirit:right"/>
        </xsl:call-template>
      </xsl:if>
    </xsl:for-each>

    project_close
</xsl:template>
```

Figura 50 – Excerto do XSLT para geração de ficheiros TCL

4.3.4. Método para Aplicação de XSLT

Após implementados os ficheiros XSLT para transformação dos documentos IP-XACT, o processo de transformação implica a criação de um objeto *XslCompiledTranform* do lado da *framework*, para fazer o carregamento da *stylesheet* e de um *XPathDocument*, para carregar o documento XML que se pretende transformar. Caso a *stylesheet* utilizada corresponda a um documento XSLT 2.0 o processo de aplicação da transformação é um pouco mais complexo. Neste caso é necessário instanciar um objeto do tipo *Processor*, que permite depois instanciar mais dois objetos, um *XdmNode* para carregar a raiz do documento XML a transformar e um *XsltTranformer* para carregar o XSLT a aplicar. De seguida é necessário passar a raiz do documento XML à *stylesheet* carregada. Antes de se proceder à transformação é ainda necessário a criação de um *Serializer* que recebe o ficheiro de *output* pretendido. A transformação pode depois ser executada através do método *run* do objeto *XsltTransformer*. A Figura 51 ilustra o processo de invocação de um XSLT 1.0 para transformação de um documento componente IP-XACT para código Verilog.

```
Directory.CreateDirectory(dirPath);
string outputName = dirPath + "/" + textBox1.Text + ".v";

XPathDocument ipxact = new XPathDocument(filePath);
XslCompiledTransform transform = new XslCompiledTransform();
string path = "../../XSLTFile1.xslt";
transform.Load(path);
XmlTextWriter file = new XmlTextWriter(outputName, null);
transform.Transform(ipxact, null, file);
file.Close();
MessageBox.Show("File created");
```

Figura 51 - Processo de Transformação através de um XSLT 1.0

4.4. Invocação de Ferramentas Externas

Nas secções anteriores descreveu-se o processo de implementação das transformações dos documentos IP-XACT para código HDL. A presente secção tem por objetivo apresentar a implementação do processo seguinte à geração de código, isto é, a compilação dos ficheiros gerados automaticamente por forma a obter código possível de ser programado na plataforma alvo.

O processo de compilação do código HDL é externo à *framework* e levado a cabo pelas ferramentas de desenvolvimento disponibilizadas por cada um dos distribuidores de FPGA. Na presente dissertação as plataformas de teste pertencem aos distribuidores Xilinx e Altera. Por este motivo a instalação das ferramentas de desenvolvimento dos mesmos é necessária para que a geração do código e programação das FPGA seja possível. No caso da Xilinx é necessária a instalação do ISE. Para a Altera o *software* a instalar é o Quartus II.

A análise realizada à interface da linha de comandos de ambas as ferramentas permitiu identificar a sequência de instruções necessárias para cada um dos casos. Na Tabela 5 são apresentadas as instruções a executar, com os campos que variam consoante o projeto a serem escritos a itálico.

Tabela 5 - Comandos para geração de *bitstream* e programação das FPGA

Xilinx

➔ run –ifn *nome_ficheiro.v* –ifmt Verilog –ofn *nome_ficheiro_output* –p *FPGA_alvo* –opt_mode speed –opt_level 1 | xst > CompiledLogs\synthesisLog.txt

➔ ngdbuild –p *FPGA_alvo* –uc *nome_ficheiro.ucf nome_ficheiro.ngc* > CompiledLogs\netlisterLog.txt

➔ map –detail –pr b *nome_ficheiro.ngd* > CompiledLogs\mapLog.txt

➔ par –w *nome_ficheiro.ncd* parout.ncd *nome_ficheiro.pcf* > CompiledLogs\parLog.txt

➔ bitgen –w –g StartUpCLK:JTAGCLK –g CRC:Enable parout.ncd *nome_ficheiro.bit nome_ficheiro.pcf* > CompiledLogs\bitGenLog.txt

➔ djtgcfg enum > CompiledLogs\devicesLog.txt

➔ djtgcfg prog –d *nome_FPGA* –i *índice_JTAG* –f *nome_ficheiro.bit*

Altera

➔ quartus_sh –t *nome_script.tcl*

➔ quartus_map *nome_projeto* > CompiledLogs\synthesisLog.txt

➔ quartus_fit *nome_projeto* > CompiledLogs\parLog.txt

➔ quartus_asm *nome_projeto* > CompiledLogs\bitGenLog.txt

➔ quartus_pgm –c usb-blaster –m jtag –o p;*nome_ficheiro.sof*

Como se pode verificar pela análise da Tabela 5, todos os comandos do *design flow* terminam com o redireccionamento do *output* para um ficheiro de texto com o nome da respetiva fase, num diretório com o nome *CompiledLogs*, criado para guardar o histórico das operações de compilação do projeto. No momento de compilação, assim que um deles está disponível, a *framework* faz o seu display dando assim um feedback ao utilizador das operações que estão em curso e quais as fases já executadas.

O processo de invocação das ferramentas externas foi dividido em duas fases para ambos os distribuidores. A primeira fase corresponde à geração do *bitstream* para a FPGA. No final desta fase o utilizador pode analisar o feedback disponibilizado pelo *framework*, ou então analisar os ficheiros na pasta *CompiledLogs* e retificar qualquer erro que possa ter ocorrido na fase de desenvolvimento do projeto. A segunda fase envolve a seleção de um dos dispositivos disponíveis para serem programados e a programação do mesmo.

Esta divisão foi implementada recorrendo à geração de dois ficheiros *batch* distintos. Um com os comandos das diversas fases de *design flow* e o outro com o comando responsável pela programação da FPGA. Dependendo do distribuidor selecionado o conjunto de ficheiros *batch* a gerar também é diferente. Assim, no final, temos dois conjuntos de ficheiros *batch*, um para a Xilinx e outro para a Altera, cada um constituído por dois ficheiros *batch*. Na Figura 52 é apresentado o excerto de código para a geração do ficheiro *batch* das fases de *design flow* para a Altera.

```
batchName = dirPath + "\\" + "bitGenerator.bat";
StreamWriter batch = new StreamWriter(batchName, false);
string device = comboBox3.Text.ToLower();

batch.WriteLine(@"cd " + dirPath);
batch.WriteLine("mkdir CompiledLogs");

batch.WriteLine("quartus_sh -t script.tcl");

batch.WriteLine("quartus_map " + textBox1.Text + @" >
CompiledLogs\synthesisLog.txt");

batch.WriteLine("quartus_fit " + textBox1.Text + @" >
CompiledLogs\parLog.txt");

batch.WriteLine("quartus_asm " + textBox1.Text + @" >
CompiledLogs\bitGenLog.txt");

batch.Close();
```

Figura 52- Geração de ficheiro *batch*

Capítulo 5

TESTES E RESULTADOS

O capítulo cinco encontra-se dividido em dois subcapítulos. O primeiro subcapítulo apresenta os testes de unidade realizados ao sistema. Tal como os dois capítulos anteriores, os testes de unidades encontram-se divididos por módulos. Por fim, no segundo subcapítulo é apresentado um teste de integração que ilustra o sistema final em funcionamento. Neste teste percorrem-se todas as fases de *design flow* embutidas na framework desenvolvida para o desenvolvimento de sistemas baseados em FPGA.

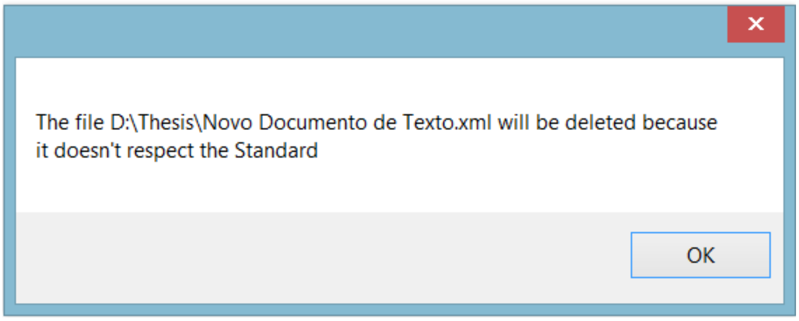
5.1. Testes de Unidade

Neste subcapítulo serão apresentados os resultados obtidos após a implementação de cada um dos módulos desenvolvidos. Os resultados aqui apresentados são referentes aos módulos individuais, sem que exista nenhuma integração entre eles. Para cada um dos testes apresenta-se uma breve descrição do teste desenvolvido e de seguida os resultados obtidos, através de imagens e tabelas.

5.1.1. Gestor de Repositório

Por forma a testar o correto funcionamento do gestor de repositório um conjunto de testes foram executados no final da fase de implementação. Assim, os testes aos quais o gestor do repositório foi submetido são:

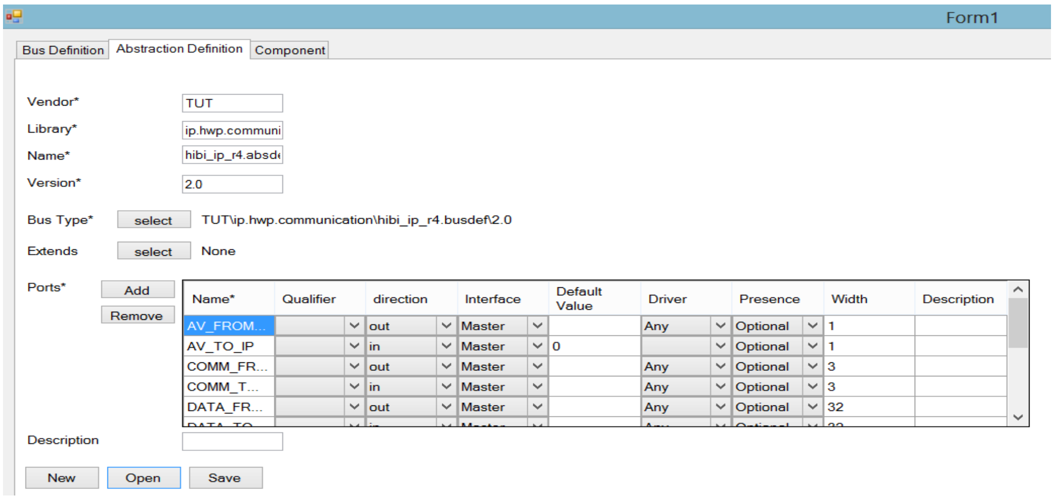
1) Executar o gestor com ficheiros IP-XACT válidos e incorretos no repositório e verificar se no final do processo de carregamento do programa os ficheiros incorretos foram eliminados. Este procedimento permite testar dois aspetos em simultâneo. Primeiro testa a capacidade do gestor de manter o repositório íntegro. Segundo, testa a correta leitura e validação dos ficheiros. O resultado obtido pode ser visualizado na Figura 53.



The file D:\Thesis\Novo Documento de Texto.xml will be deleted because it doesn't respect the Standard

Figura 53 - Mensagem de Ficheiro IP-XACT Inválido

2) Apesar do teste anterior permitir testar o processo de leitura do ficheiro, deve ser executado um teste onde o *output* dado pelo programa são os dados extraídos dos ficheiros IP-XACT do repositório. Os ficheiros IP-XACT a testar deverão ser providenciados por um programa já presente no mercado, como por exemplo o Kactus2. A informação dada pela *framework* deve corresponder a toda a informação presente no documento IP-XACT. Na Figura 54 é apresentado o resultado obtido do processo de leitura de um documento IP-XACT do tipo *abstractionDefinition*, enquanto na Figura 55 é apresentado o documento original. Como se pode constatar através da comparação das duas figuras, o processo de leitura apresenta com sucesso os dados do documento.



Vendor* TUT
Library* ip.hwp.communi
Name* hibi_ip_r4.absd
Version* 2.0
Bus Type* TUT\ip.hwp.communication\hibi_ip_r4.busdef\2.0
Extends None
Ports*

Name*	Qualifier	direction	Interface	Default Value	Driver	Presence	Width	Description
AV_FROM...		out	Master		Any	Optional	1	
AV_TO_IP		in	Master	0		Optional	1	
COMM_FR...		out	Master		Any	Optional	3	
COMM_T...		in	Master		Any	Optional	3	
DATA_FR...		out	Master		Any	Optional	32	

Description

Figura 54 - *Output* do processo de Leitura

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Created by Kactus 2 document generator 10:36:13 to syys 29
2011-->
<spirit:abstractionDefinition ...>
    ...
    <spirit:busType spirit:vendor="TUT"
spirit:library="ip.hwp.communication"
spirit:name="hibi_ip_r4.busdef" spirit:version="2.0"/>
    <spirit:ports>
        <spirit:port>
            <spirit:logicalName>AV_FROM_IP</spirit:logicalName>
            <spirit:wire>
                <spirit:onMaster>
                    <spirit:presence>optional</spirit:presence>
                    <spirit:width>1</spirit:width>
                    <spirit:direction>out</spirit:direction>
                </spirit:onMaster>
                <spirit:requiresDriver
spirit:driverType="any"/>false</spirit:wire>
        </spirit:port>
        <spirit:port>
            <spirit:logicalName>AV_TO_IP</spirit:logicalName>
            <spirit:wire>
                <spirit:onMaster>
                    <spirit:presence>optional</spirit:presence>
                    <spirit:width>1</spirit:width>
                    <spirit:direction>in</spirit:direction>
                </spirit:onMaster>
                <spirit:defaultValue>0</spirit:defaultValue>
      ...
```

Figura 55 - Excerto do ficheiro IP-XACT lido

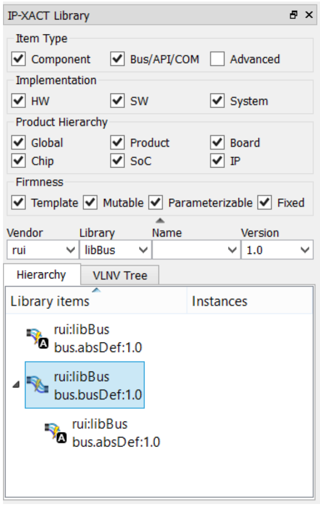
3) Por fim deve ser executado um teste ao processo de escrita de ficheiros IP-XACT. Para validar este teste devemos analisar os ficheiros gerados e verificar se cumprem todas as especificações do *standard*. Os ficheiros devem ser colocados na biblioteca de um programa compatível com IP-XACT. Se o programa considerar os ficheiros como sendo válidos, pode-se assumir que o processo de escrita está correto. Na Figura 56 e na Figura 57 são apresentados dois documentos IP-XACT gerados pelo gestor do repositório. A Figura 58 e a Figura 59 são parte da *framework* Kactus2 onde se pode verificar a inexistência de irregularidades nos documentos gerados, parte superior da figura, e a correta leitura dos dados dos documentos, parte inferior da figura.

```xml
<?xml version="1.0" encoding="utf-8"?>
<spirit:abstractionDefinition xmlns:spirit="coisa">
  <spirit:vendor>rui</spirit:vendor>
  <spirit:library>libBus</spirit:library>
  <spirit:name>bus.absDef</spirit:name>
  <spirit:version>1.0</spirit:version>
  <spirit:busType spirit:vendor="rui" spirit:library="libBus"
spirit:name="bus.busDef" spirit:version="1.0" />
  <spirit:ports>
    <spirit:port>
      <spirit:logicalName>port0</spirit:logicalName>
      <spirit:displayName>port0</spirit:displayName>
      <spirit:wire>
        <spirit:qualifier>
          <spirit:isClock>true</spirit:isClock>
        </spirit:qualifier>
        <spirit:onMaster>
          <spirit:presence>required</spirit:presence>
          <spirit:width>2</spirit:width>
          <spirit:direction>in</spirit:direction>
        </spirit:onMaster>
        ...
        <spirit:requiresDriver
spirit:driverType="none">true</spirit:requiresDriver>
      </spirit:wire>
    </spirit:port>
  </spirit:ports>
</spirit:abstractionDefinition>
```

Figura 56 - *abstarctionDefinition* gerado

```xml
<?xml version="1.0" encoding="utf-8"?>
<spirit:busDefinition xmlns:spirit="coisa">
  <spirit:vendor>rui</spirit:vendor>
  <spirit:library>libBus</spirit:library>
  <spirit:name>bus.busDef</spirit:name>
  <spirit:version>1.0</spirit:version>
  <spirit:directConnection>false</spirit:directConnection>
  <spirit:isAddressable>false</spirit:isAddressable>
</spirit:busDefinition>
```

Figura 57 - *busDefinition* gerado


IP-XACT Library

Item Type
☑ Component ☑ Bus/API/COM ☐ Advanced

Implementation
☑ HW ☑ SW ☑ System

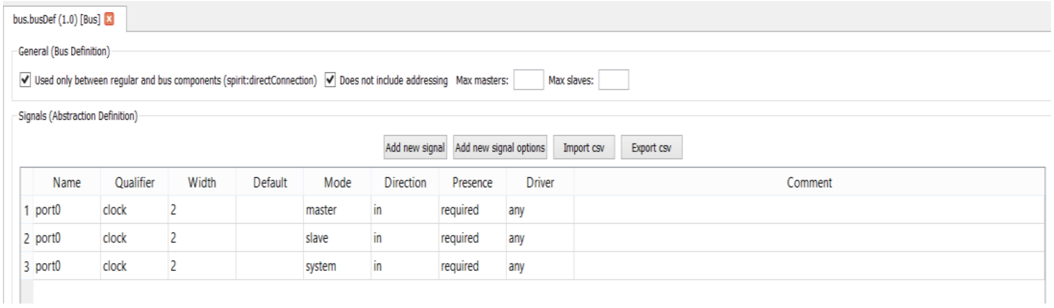
Product Hierarchy
☑ Global ☑ Product ☑ Board
☑ Chip ☑ SoC ☑ IP

Firmness
☑ Template ☑ Mutable ☑ Parameterizable ☑ Fixed

Vendor: rui
Library: libBus
Name:
Version: 1.0

Library items	Instances
rui:libBus bus.absDef:1.0	
rui:libBus bus.busDef:1.0	
rui:libBus bus.absDef:1.0	

Figura 58 - Prova de ausência de irregularidades nos documentos gerados (Kactus2)


bus.busDef (1.0) [Bus]

General (Bus Definition)
☑ Used only between regular and bus components (spirit:directConnection) ☑ Does not include addressing Max masters: Max slaves:

Signals (Abstraction Definition)

	Name	Qualifier	Width	Default	Mode	Direction	Presence	Driver	Comment
1	port0	clock	2		master	in	required	any	
2	port0	clock	2		slave	in	required	any	
3	port0	clock	2		system	in	required	any	

Figura 59 - Leitura dos documentos gerados (Kactus2)

5.1.2. *Framework* (GUI)

Os testes realizados à *framework* incidiram na componente de desenho de *designs*. As restantes funcionalidades estão relacionadas com a interface entre módulos e por esse motivo foram testadas aquando a integração do sistema final (ver capítulo 5.2).

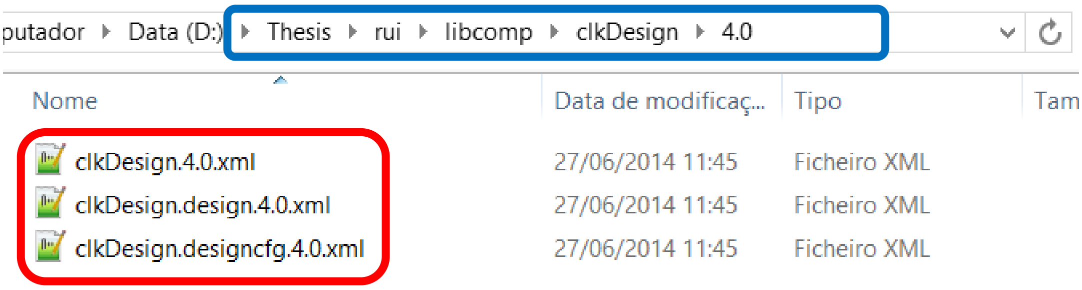
O principal aspeto a testar no momento de criação de *designs* é a instanciação de IPs. Só é possível realizar as verificações e validações envolvidas no processo de criação de sistemas se a informação presente no documento IP-XACT do IP instanciado se encontrar correta. A correta instanciação de um IP pode ser verificada através da comparação da Figura 60, que ilustra um IP instanciado na área de desenho da *framework*, e a Figura 61, onde se pode ver um excerto do documento IP-XACT do IP instanciado. Como é possível visualizar, o IP possui duas interfaces, com o nome *clk* e *ovfFlag*, que se encontram instanciadas no documento IP-XACT no campo *busInterfaces*. Tanto o número como o nome das interfaces se encontram corretos assim como o tipo do IP. Apesar de não ser visível no objeto gráfico do IP, as informações carregadas no momento da instanciação do IP, vão muito além do nome e número de interfaces. Na realidade toda a informação presente no documento IP-XACT é carregada para permitir todo o tipo de verificações e validação ao sistema desenhado.

Figura 60 - IP instanciado

```
<spirit:component
xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5"
xmlns:esrg="Uminho">
  <spirit:vendor>rui</spirit:vendor>
  <spirit:library>libcomp</spirit:library>
  <spirit:name>timer1bit</spirit:name>
  <spirit:version>1.0</spirit:version>
  <spirit:busInterfaces>
    <spirit:busInterface>
      <spirit:name>clk</spirit:name>
      ...
    </spirit:busInterface>
    <spirit:busInterface>
      <spirit:name>ovfFlag</spirit:name>
      ...
    </spirit:busInterface>
  </spirit:busInterfaces>
</spirit:component>
```

Figura 61 - Documento IP-XACT do IP instanciado

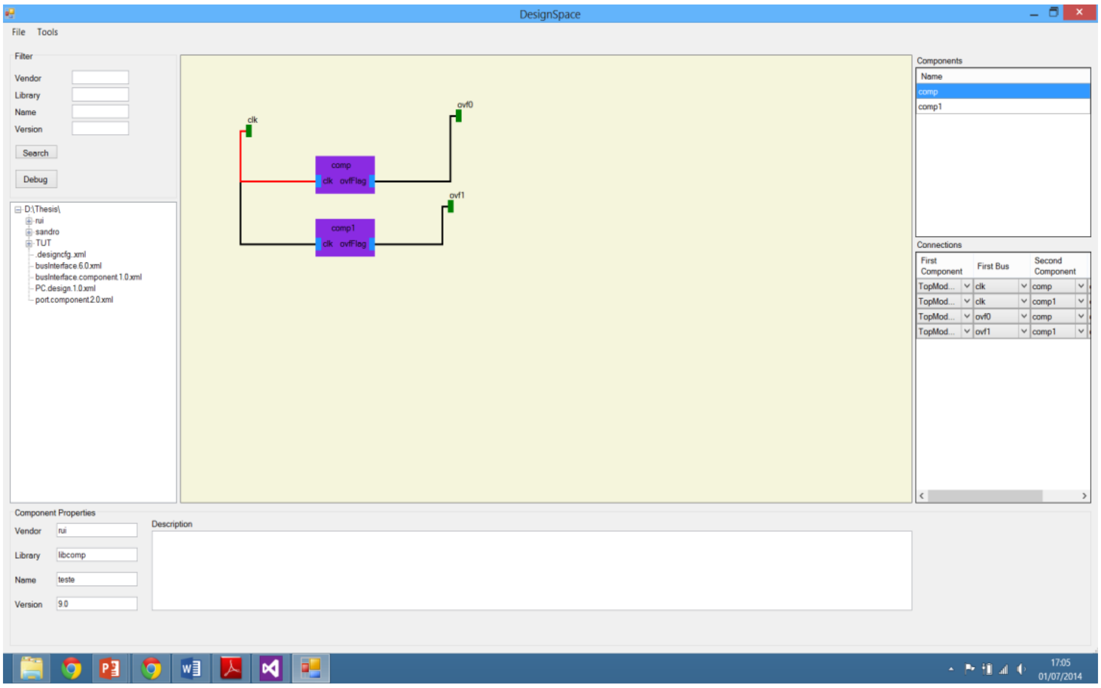
Depois de concluído o desenho do sistema é importante testar o processo de criação dos documentos relacionados com o *design* desenvolvido. Para além do ficheiro de *design* o sistema deve gerar automaticamente um documento de configuração de *design*, do tipo *designConfiguration*, e o documento do componente de topo, do tipo *component*. A Figura 62 demonstra os documentos gerados, respeitando a convenção de nomes definida anteriormente nesta dissertação.



Nome	Data de modificaç...	Tipo	Tam
clkDesign.4.0.xml	27/06/2014 11:45	Ficheiro XML	
clkDesign.design.4.0.xml	27/06/2014 11:45	Ficheiro XML	
clkDesign.designcfg.4.0.xml	27/06/2014 11:45	Ficheiro XML	

Figura 62 - Documentos gerados a partir de um *Design*

Por fim, na Figura 63 é apresentado o aspeto da *framework* desenvolvida para a criação de *designs* compatíveis com o *standard* IP-XACT. Na parte superior da janela encontram-se as opções para criar, guardar e carregar *designs*. À esquerda encontram-se duas tabelas, uma onde se podem consultar os IPs instanciados no *design* atual, a outra com as conexões existentes, sendo possível alterar uma conexão ou criar uma nova a partir de um duplo clique sobre a tabela. Na parte inferior é apresentada a informação relativa à VLNV e à descrição do *design* e no lado direito da janela um filtro para pesquisa de IPs e uma lista com os IPs presentes no repositório.



First Component	First Bus	Second Component
TopMod...	clk	comp
TopMod...	clk	comp1
TopMod...	ovf0	comp
TopMod...	ovf1	comp1

Vendor: rui
Library: libcomp
Name: teste
Version: 9.0

Figura 63 - Aspeto gráfico da *Framework*

5.1.3. Gerador de Código

O módulo responsável pela geração de código assenta essencialmente nas *stylesheets* de transformação e ficheiros *batch*, responsáveis por invocar as ferramentas externas. Desta forma, importa testar o processo de transformação levada a cabo por cada uma das *stylesheets*, o processo de geração de código a partir da invocação das ferramentas externas através da linha de comandos e o processo de programação das diferentes plataformas alvo, no caso da presente dissertação a Basys2 e a Cyclone II.

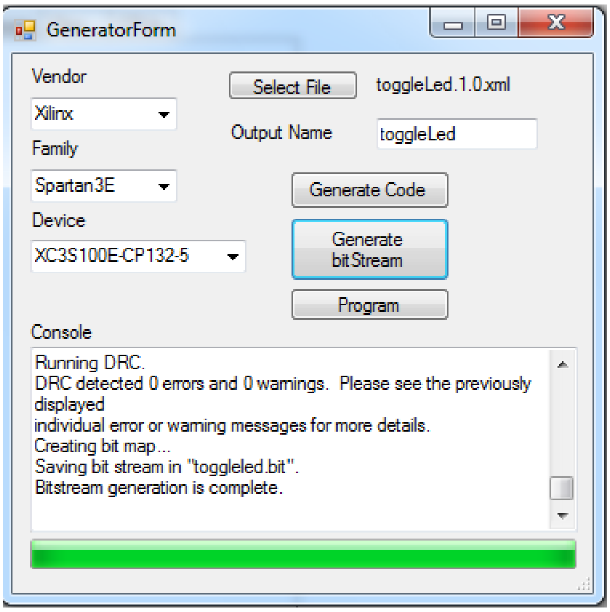
Para testar o processo de transformação, o autor criou um documento IP-XACT (Figura 64), recorrendo ao módulo de gestão de repositório desta dissertação, já devidamente testado e validado. Ao ficheiro gerado foram aplicados os três diferentes *stylesheets* desenvolvidos, um para gerar o código HDL, outro para gerar o ficheiro UCF para as plataformas da Xilinx e outro para o ficheiro TCL para criar o projeto para as plataformas da Altera.

```
<?xml version="1.0" encoding="utf-8"?>
<spirit:component xmlns:spirit="http://www.spiritco
  <spirit:vendor>rui</spirit:vendor>
  <spirit:library>libComp</spirit:library>
  <spirit:name>toggleLed</spirit:name>
  <spirit:version>1.0</spirit:version>
  <spirit:model>
    <spirit:ports>
      <spirit:port>
        <spirit:name>led</spirit:name>
        <spirit:wire>
          <spirit:direction>out</spirit:direction>
        </spirit:wire>
      </spirit:port>
      <spirit:port>
        <spirit:name>clk</spirit:name>
        <spirit:wire>
          <spirit:direction>in</spirit:direction>
        </spirit:wire>
      </spirit:port>
      <spirit:port>
        <spirit:name>rst</spirit:name>
        <spirit:wire>
          <spirit:direction>in</spirit:direction>
        </spirit:wire>
      </spirit:port>
    </spirit:ports>
  </spirit:model>
</spirit:component>
```

Figura 64 - Ficheiro XML criado para Testes

O interface criado para o terceiro módulo é apresentado na Figura 65. Utilizando esta interface pode-se selecionar o documento IP-XACT a transformar, o distribuidor, família e dispositivo FPGA e por fim o nome desejado para os ficheiros resultantes da transformação.

O botão com o texto *Generate Code* aplica as *stylesheets* criadas ao documento selecionado e gere também os ficheiros *batch* para a invocação das ferramentas externas. Após concluída esta fase, é possível gerar o respetivo *bitstream* para a plataforma selecionada carregando no botão com o texto *Generate Bitstream*. Esta operação cria também uma lista com todos os dispositivos FPGA disponíveis. O feedback produzido pelas ferramentas externas é dado ao utilizador através da caixa de texto situada na parte inferior da interface. Ao carregar no botão com o texto *Program* um *pop-up* com a lista de *FPGA* disponíveis é criado, onde o utilizador pode selecionar qual o dispositivo que pretende programar. Ao carregar no botão *OK* desse *pop-up* é gerado e executado o ficheiro *batch* com o comando para programação da FPGA selecionada.

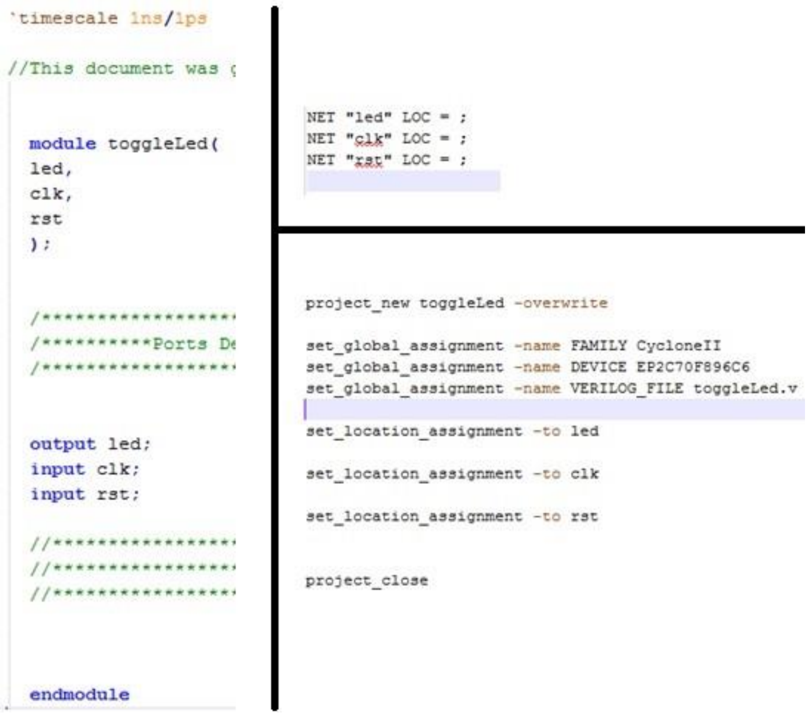


Vendor: Xilinx
Family: Spartan3E
Device: XC3S100E-CP132-5
toggleLed.1.0.xml
Output Name: toggleLed

```
Running DRC.
DRC detected 0 errors and 0 warnings.  Please see the previously
displayed
individual error or warning messages for more details.
Creating bit map...
Saving bit stream in "toggleled.bit".
Bitstream generation is complete.
```

Figura 65 - Interface do Gerador de Código

O resultado da transformação exercida pelas diferentes *stylesheets* aplicadas é apresentado na Figura 66. À esquerda pode ver-se o código Verilog gerado, do lado direito, parte superior, o ficheiro UCF e da parte inferior o ficheiro TCL. Comparando o documento IP-XACT e as respetivas transformações pode-se constatar que todos os portos declarados no documento IP-XACT se encontram instanciados em todos os documentos gerados. Para além da correta instanciação, o tamanho dos portos e a sua direção no ficheiro HDL corresponde ao definido no documento IP-XACT.



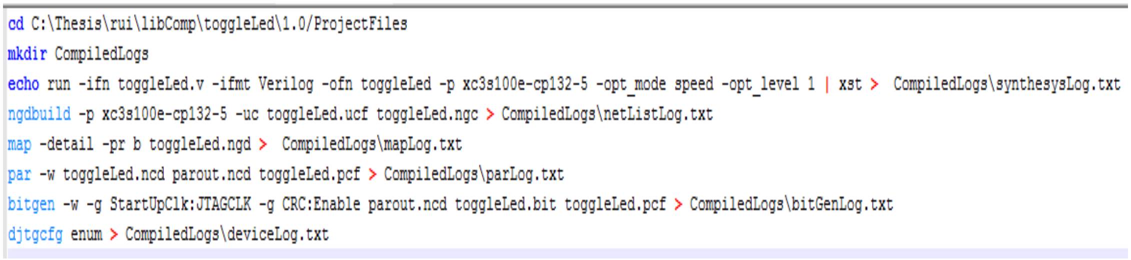
```
`timescale 1ns/1ps

//This document was

module toggleLed(
led,
clk,
rst
);

/*****************
/**********Ports De
/*****************

output led;
input clk;
input rst;

//****************
//****************
//****************

endmodule
```

```
NET "led" LOC = ;
NET "clk" LOC = ;
NET "rst" LOC = ;
```

```
project_new toggleLed -overwrite

set_global_assignment -name FAMILY CycloneII
set_global_assignment -name DEVICE EP2C70F896C6
set_global_assignment -name VERILOG_FILE toggleLed.v

set_location_assignment -to led

set_location_assignment -to clk

set_location_assignment -to rst


project_close
```

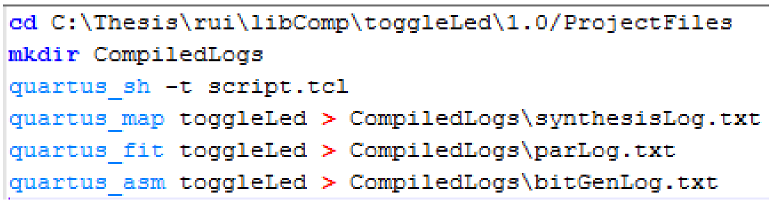
Figura 66 - Resultados do processo de transformação

Os ficheiros *batch* necessários para a invocação das ferramentas externas encontram-se expostos na Figura 67, no caso da Xilinx e na Figura 68 para o caso da Altera.



```
cd C:\Thesis\rui\libComp\toggleLed\1.0/ProjectFiles
mkdir CompiledLogs
echo run -ifn toggleLed.v -ifmt Verilog -ofn toggleLed -p xc3s100e-cp132-5 -opt_mode speed -opt_level 1 | xst >  CompiledLogs\synthesysLog.txt
ngdbuild -p xc3s100e-cp132-5 -uc toggleLed.ucf toggleLed.ngc > CompiledLogs\netListLog.txt
map -detail -pr b toggleLed.ngd >  CompiledLogs\mapLog.txt
par -w toggleLed.ncd parout.ncd toggleLed.pcf > CompiledLogs\parLog.txt
bitgen -w -g StartUpClk:JTAGCLK -g CRC:Enable parout.ncd toggleLed.bit toggleLed.pcf > CompiledLogs\bitGenLog.txt
djtgcfg enum > CompiledLogs\deviceLog.txt
```

Figura 67 - Ficheiro *batch* para o ISE da Xilinx



```
cd C:\Thesis\rui\libComp\toggleLed\1.0/ProjectFiles
mkdir CompiledLogs
quartus_sh -t script.tcl
quartus_map toggleLed > CompiledLogs\synthesisLog.txt
quartus_fit toggleLed > CompiledLogs\parLog.txt
quartus_asm toggleLed > CompiledLogs\bitGenLog.txt
```

Figura 68 - Ficheiro *batch* para o *software* Quartus II

Comparando os dados presentes na Figura 65 com o ficheiro *batch* da Figura 67 pode-se observar que o ficheiro *batch* é o resultado do processo de geração de código para o caso apresentado pela Figura 65. Ao selecionar-se o distribuidor Xilinx, o ficheiro *batch* gerado foi o da Figura 67 e não o da Figura 68, como era esperado. O valor associado à opção –p corresponde ao dispositivo selecionado na interface gráfica, assim como o nome para os ficheiros gerados corresponde ao nome presente na caixa de texto com a legenda *Output Name*. Comparando agora o mesmo ficheiro *batch* com os comandos necessários para a geração do *bitstream*, apresentados no capítulo 4.4, pode-se verificar que os comandos se encontram bem construídos. Se se comparar o ficheiro *batch* da Figura 68, que é gerado caso a opção *vendor* do interface gráfico seja Altera, com a tabela apresentada no capítulo 4.4, conclui-se que os comandos do ficheiro também estão construídos da forma correta. O nome utilizado no campo que define o projeto a compilar corresponde ao texto presente na caixa de texto com a legenda *Output Name*, tal como se pretendia, uma vez que é o valor desta caixa de texto que define o nome do projeto no ficheiro TCL, apresentado na Figura 66, através da linha *project_new*. É também neste ficheiro TCL que as opções relacionadas com o projeto se encontram definidas como por exemplo a família, dispositivo e ficheiros a incluir no projeto. Estas opções podem ser identificadas na Figura 66 através da palavra reservada *set_global_assignment*. Estes valores correspondem aos valores definidos no interface gráfico.

Pode-se então concluir que o módulo de geração de código cumpre todas as especificações pretendidas. A geração de código é realizada da forma correta e os dados presentes nos ficheiros gerados correspondem aos definidos pelo utilizador e presentes nos documentos IP-XACT. A programação das diferentes plataformas foi também executada com sucesso como se pode visualizar na Figura 69.

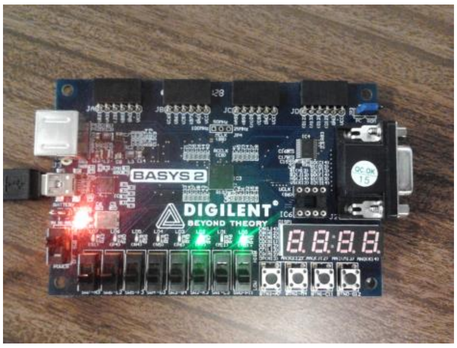


Figura 69 - Programa em execução sobre um sistema implementado na Basys2

Na presente secção foram explicados os procedimentos de teste e os resultados obtidos para o módulo de geração de código. Concluiu-se, pela análise dos resultados obtidos, que o correto funcionamento do módulo está assegurado e que todas as especificações definidas no capítulo 3.1.3 e 3.3 foram respeitadas. Estando já validados os restantes módulos nos capítulos 5.1.1 e 5.1.2, na secção seguinte realiza-se um teste de integração por forma a validar o correto funcionamento da *framework* desenvolvida.

5.2. Teste de Integração

O teste de integração desenvolvido explorou todas as funcionalidades oferecidas pela *framework* desenvolvida, desde a criação de barramentos e componentes simples, até à criação de sistemas e a geração automática do código HDL associado. Para varrer todo o conjunto de funções optou-se por criar um bloco com dois timers a partir de um timer simples. O teste iniciou-se com o desenvolvimento das definições (*busDefinition* e *abstractionDefinition*) para o sinal de relógio e de *overflow*. De seguida, criou-se um componente simples com o nome *timer* com duas interfaces, uma do tipo do sinal de relógio e outra do tipo *overflow*, e gerou-se o código HDL respetivo. Após concluído o processo de geração do HDL, os ficheiros foram alterados para acrescentar o comportamento desejado ao componente. Feito isto, o processo de criação do componente simples ficou completo.

Passou-se então para a criação do sistema hierárquico do bloco de timers. A partir do mecanismo de *drag-and-drop* instanciou-se dois componentes do tipo do timer anteriormente criado, aos quais se deu o nome de *timer0* e *timer1*. Em seguida criaram-se três interfaces para o módulo de topo, através da interface disponível no clique do botão direito do rato, uma para o sinal de relógio e outras duas para os sinais de *overflow* provenientes dos timers instanciados. Concluída a criação do sistema, procedeu-se à geração dos documentos IP-XACT respetivos através da opção *New*.

O passo seguinte envolveu a geração automática do ficheiro HDL para o *design* desenvolvido. Desta vez, após concluída esta fase não foi necessário a alteração do código Verilog para definição do comportamento porque não se pretendia que o módulo de topo efetuação qualquer processamento. O passo seguinte envolveu a criação do *bitstream* para a plataforma de desenvolvimento Basys2 e a respetiva programação da placa. De salientar que, apesar do teste ser efetuado para a Basys2, o sistema desenvolvido podia ser programado numa outra FPGA sem ser necessário qualquer alteração no *design*, bastando selecionar um gerador para a FPGA desejada, ou seja, o *design* desenvolvido é agnóstico da plataforma. O teste descrito pode ser visualizado através do *link* http://youtu.be/uk_tSrGx6-c.

Capítulo 6

CONCLUSÃO

Apresentados todos os passos relacionados com o desenvolvimento do projeto proposto pela dissertação, neste último capítulo são apresentadas todas as ilações retiradas. É também proposto um conjunto de sugestões para melhorar o sistema desenvolvido, por forma a guiar o trabalho futuro.

6.1. Conclusão

Com o aumento da complexidade dos sistemas atuais a reutilização de IPs, quer de *hardware* quer de *software*, é um *must*. A presente dissertação foi desenvolvida para explorar essa reutilização e o seu impacto no tempo de desenvolvimento de um sistema complexo. É ainda abordada a interoperabilidade entre ferramentas através da utilização de ferramentas externas e da possibilidade de configuração das mesmas dentro da *framework* desenvolvida.

A dissertação apresenta o processo de desenvolvimento de uma *framework* compatível com o *standard* 1685 do IEEE IP-XACT. Para diminuir a complexidade de implementação, foi realizada uma divisão do projeto em três módulos distintos. No final do desenvolvimento de todos os módulos procedeu-se à integração do sistema final utilizando o módulo da *framework* como ponto de ligação dos restantes módulos.

Os objetivos inicialmente propostos foram devidamente cumpridos, conseguindo-se uma *framework* que cobre todas as etapas envolvidas no *design flow* para o desenvolvimento de sistemas baseados em FPGA. Inicialmente realizou-se uma análise a vários ambientes de desenvolvimento de *software* que permitissem o desenvolvimento de outras *frameworks*. A escolha recaiu sobre a utilização do Visual Studio 2012 por diminuir a relação tempo/esforço no processo de desenvolvimento. De seguida efetuou-se em estudo à ferramenta Kactus2 com o objetivo de realizar um levantamento das funcionalidades básicas a disponibilizar. A ferramenta Kactus2 foi mais tarde utilizada para efetuar os testes aos documentos IP-XACT gerados pela *framework* implementada. Todos os testes realizados aos documentos gerados obtiveram resultado positivo garantindo assim que o suporte ao *standard* foi cumprido. Concluído suporte à gestão do repositório, implementaram-se os geradores de código para criação de ficheiros HDL a partir de documentos IP-XACT. Mais uma vez, utilizaram-se os documentos gerados pelo Kactus2 em paralelo com os documentos gerados pela *framework* desenvolvida para testar a correta geração dos documentos HDL. Por fim desenvolveu-se uma interface com o utilizador para permitir a criação de *designs* e realizar a interface entre os vários módulos. A fim de testar todo o sistema, programaram-se duas plataformas distintas a partir do mesmo *design*, alterando-se apenas o gerador selecionado, comprovando assim que todos os objetivos propostos foram cumpridos.

A realização desta dissertação contribuiu fortemente no conhecimento do autor, bem como, na comunidade científica através da coautoria de capítulos de livro em diversos campos de investigação nomeadamente na área das FPGA e dos sistemas operativos embebidos. De seguida serão apresentadas as publicações in press ou já publicadas.

J. Pereira, D. Oliveira, P. Matos, R. Machado, S. Pinto, T. Gomes, V. Silva, E. Qaralleh, N. Cardoso, P. Cardoso, “Hardware-assisted Real-Time Operating System Deployed on FPGA”, to be published on "Informatik/Kommunikationstechnik" subseries of the "Fortschritt-Berichte VDI" series edited by VDI Verlag, 2014

6.2. Trabalho Futuro

Apesar de todos os objetivos propostos terem sido alcançados, existe um conjunto alargado de funcionalidades que podem ser implementadas sobre a *framework* desenvolvida, de forma a expandir e melhorar a interface com o utilizador.

Como referido no capítulo 4.2.1, a construção de *designs* na *framework* é realizada recorrendo a uma abordagem *bottom-up*. Para fornecer uma maior flexibilidade de construção ao utilizador, propõe-se a implementação de mecanismos baseados em abordagens *top-down*, combinados com a abordagem *bottom-up* já existente, o que certamente acrescentaria maior valor à *framework*. Para tal seria necessário o desenvolvimento de interfaces que permitissem a criação de IPs cujos barramentos e respetiva composição seria definida ao longo do desenvolvimento do restante sistema.

A segunda proposta de implementação consiste no desenvolvimento de um simulador para os *designs* criados. Apesar da *framework* atual impedir que se realizem conexões entre dois barramentos com interfaces diferentes, é possível a existência de erros comportamentais em alguns IPs, pelo que esta funcionalidade seria bastante útil como complemento ao processo de co-validação e co-verificação.

A edição de código disponibilizada pela *framework* é realizada através do programa notepad++ ou, caso este *software* não se encontre instalado, através de um editor de texto muito simples baseado no bloco de notas do Windows. Para melhorar a experiencia do utilizador e facilitar o *co-design* de *hardware* e *software*, propõe-se o desenvolvimento de um editor de texto com coloração de palavras-chave, completação de código e deteção de erros.

Relativamente *standard* IP-XACT, Fornecidos os mecanismos de gestão, implementados pela *framework* desenvolvida, é fundamental o desenvolvimento de extensões de *software* para o *standard* por forma a permitir a associação de IPs de *software* a IPs de *hardware* no momento do desenvolvimento de um *design*.

Por último sugeria-se dotar a framework da capacidade para a exploração do espaço de soluções (DSE - *Design Space Exploration*), particularmente útil no desenvolvimento de sistemas complexos.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] E. Girczyc and S. Carlson, “Increasing design quality and engineering productivity through design reuse,” *Proc. 30th Int. Des. ...*, pp. 48–53, 1993.

[2] 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: 05-Oct-2013].

[3] A. de Melo and H. Barringer, “A foundation for formal reuse of hardware,” *Correct Hardw. Des. Verif. ...*, pp. 124–145, 1995.

[4] W. Kruijtzer, P. van der Wolf, E. de Kock, J. Stuyt, W. Ecker, a. Mayer, S. Hustin, C. Amerijckx, S. de Paoli, and E. Vaumorin, “Industrial IP integration flows based on IP-XACT standards,” *2008 Des. Autom. Test Eur.*, pp. 32–37, Mar. 2008.

[5] IEEE Computer Society, “IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows,” 2010.

[6] Accellera, “IP-XACT Working Group.” [Online]. Available: http://www.accellera.org/activities/committees/ip-xact/. [Accessed: 10-Oct-2013].

[7] J. Swanson, “An XACT science,” *newelectronics*, no. February, pp. 27–28, 2013.

[8] G. Kaur, “DIGITAL DESIGN FLOW,” in *VHDL: Basics to Programming*, 2011.

[9] D. Chen, “Design Automation for Microelectronics, Springer Handbook of Automation,” in *icims.csl.uiuc.edu*, 2009.

[10] P. Schaumont, “Hardware/software co-design is a starting point in embedded systems architecture education,” *Proc. WESE*, 2008.

[11] Altera Corporation, “AN 311 : Standard Cell ASIC to FPGA Design,” no. April. pp. 1–28, 2009.

[12] H. B. Kommuru and H. Mahmoodi, “Overview of ASIC Flow,” in *ASIC Design Flow Tutorial Using Synopsys Tools*, 2009.

[13] J. Serrano, “Introduction to FPGA design,” *8th Work. Electron. LHC Exp.*, pp. 231–247, 2004.

[14] Xilinx, “Floorplanning Overview,” in *Floorplanning Methodology Guide*, vol. 633, 2012, pp. 1–34.

[15] L.-T. W. Y.-W. C. K.-T. (Tim) Cheng, “Placement,” in *Electronic Design Automation*, Morgan Kaufmann, 2009, p. 972.

[16] M. J. S. Smith, “Routing,” in *Application-specific integrated circuits,* Boston: Addison-Wesley Longman, 1997.

[17] Xilinx, “Design Flow,” in *Development System Reference Guide,* 2008.

[18] funbase, “Kactus2,” 2009. [Online]. Available: http://funbase.cs.tut.fi/#kactus2. [Accessed: 28-Oct-2013].

[19] E. Vaumorin, “Magillem introduces MAGILLEM 4.0, the most comprehensive Integrated Design Environment based on the IP-XACT standards by The SPIRIT Consortium^{TM},” 2007. [Online]. Available: http://www.magillem.com/eda/magillem-introduces-magillem-4-0-the-most-comprehensive-integrated-design-environment-based-on-the-ip-xact-standards-by-the-spirit-consortium. [Accessed: 09-Mar-2014].

[20] Magillem, “Magillem Product Training,” 2012. [Online]. Available: http://www.magillem.com/eda/magillem-product-training. [Accessed: 09-Mar-2014].

[21] M. Zys, E. Vaumorin, and I. Sobanski, “Straightforward IP Integration with IP-XACT RTL-TLM Switching,” 2008. [Online]. Available: http://www.design-reuse.com/articles/18558/ip-xact-rtl-tlm-switching.html. [Accessed: 09-Mar-2014].

[22] Synopsys, “Synopsys coreTools,” 2008.

[23] D. Hou and Y. Wang, “An empirical analysis of the evolution of user-visible features in an integrated development environment,” *Proc. 2009 Conf. Cent. Adv. Stud. Collab. Res. - CASCON ’09,* p. 122, 2009.

[24] M. Rouse, “integrated development environment (IDE),” 2007. [Online]. Available: http://searchsoftwarequality.techtarget.com/definition/integrated-development-environment. [Accessed: 21-Oct-2013].

[25] J. Kyrnin, “What is an IDE and Do You Need an IDE to Build Web Applications?” [Online]. Available: http://webdesign.about.com/od/webprogramming/a/what-is-an-ide.htm. [Accessed: 21-Oct-2013].

[26] G. R. Jacobs, “Top 5 Integrated Development Environments,” 2010. [Online]. Available: http://hackaday.com/2010/08/24/top-5-integrated-development-environments/. [Accessed: 21-Oct-2013].

[27] W. Weinberg, “Real Programmers Do Use IDEs.”

[28] R. Stratulat, “The importance of a good IDE,” 2005. [Online]. Available: http://www.stratulat.com/blog/the-importance-of-a-good-ide. [Accessed: 21-Oct-2013].

[29] D. W. Jones, “Punched Cards,” 2012. [Online]. Available: http://homepage.cs.uiowa.edu/~jones/cards/history.html. [Accessed: 15-Nov-2013].

[30] In.com, “Maestro I.” [Online]. Available: http://www.in.com/maestro-i/biography-180639.html. [Accessed: 15-Nov-2013].

[31] D. A. Lienhart, “SoftBench 5.0 : The Evolution of an Integrated Software Development Environment,” 1997.

[32] V. D. M. Veiga, “Ambiente Integrado de Desenvolvimento para um Sistema Operativo Tempo-Real,” Universidade do Minho, 2012.

[33] Microsoft, “Visual Studio Online,” 2013. [Online]. Available: http://www.visualstudio.com/products/visual-studio-online-overview-vs. [Accessed: 23-Oct-2013].

[34] J. Teich, “Hardware/software codesign: The past, the present, and predicting the future,” *Proc. IEEE*, vol. 100, pp. 1411–1430, 2012.

[35] Refsnes Data, “XPath Syntax,” 2014. [Online]. Available: http://www.w3schools.com/XPath/xpath_syntax.asp. [Accessed: 21-Apr-2014].

[36] H. Court, “Digilent Basys2 Board Reference Manual,” vol. 99163, no. 509. pp. 1–12, 2010.

[37] I. Digilent, “Digilent Adept,” 2014. [Online]. Available: http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,66,69&Prod=ADEPT&CFID=4622247&CFTOKEN=ce4045e4b80fc71f-597AD6F4-5056-0201-02D495EB8A9DFD73. [Accessed: 22-Apr-2014].

[38] Terasic Technologies, “Altera DE2-70 User Manual.” p. 94, 2009.

[39] Xilinx, “Command-Line Tools,” vol. 628. 2012.

[40] Xilinx, “XST User Guide.” pp. 587–600.

[41] Diligent, “djtgcfg - Digilent JTAG Config Utility,” no. 1. pp. 1–2, 2010.

[42] A. Corporation, “Quartus II Handbook Version 13 . 1 Volume 2 : Design Implementation and Optimization,” vol. 2. 2013.

[43] A. Corporation, “Quartus II Scripting Reference Manual.” 2013.

[44] Michael H. Kay, “SAXON The XSLT and XQuery Processor,” 2013. [Online]. Available: http://saxon.sourceforge.net/. [Accessed: 24-Apr-2014].

[45] Saxon, “Saxon,” 2013. [Online]. Available: http://www.saxonica.com/documentation/#!about/whatis. [Accessed: 24-Apr-2014].

[46] Microsoft, “Visual Studio,” 2013. [Online]. Available: http://www.visualstudio.com. [Accessed: 23-Oct-2013].

[47] Microsoft, “Integrated Development Environment,” 2013. [Online]. Available: http://www.visualstudio.com/en-us/explore/ide-vs.aspx. [Accessed: 23-Oct-2013].

[48] Microsoft, “Application Development.” [Online]. Available: http://www.visualstudio.com/pt-pt/explore/application-development-vs#Scenario2_1. [Accessed: 23-Oct-2013].

[49] S. Somasegar, “Visual Studio 2013 Launch: Announcing Visual Studio Online,” 2013. [Online]. Available: http://blogs.msdn.com/b/somasegar/archive/2013/11/13/visual-studio-2013-launch-announcing-visual-studio-online.aspx. [Accessed: 23-Oct-2013].

[50] Microsoft, “Lync,” 2012. [Online]. Available: http://office.microsoft.com/pt-pt/lync/. [Accessed: 23-Oct-2013].

[51] Eclipse Foundation, “About the Eclipse Foundation.” [Online]. Available: http://www.eclipse.org/org/. [Accessed: 24-Oct-2013].

[52] D. Gallardo, “Getting started with the Eclipse Platform,” 2002. [Online]. Available: http://www.ibm.com/developerworks/library/os-ecov/. [Accessed: 24-Oct-2013].

[53] Eclipse Foundation, “Eclipse Documentation.” [Online]. Available: http://help.eclipse.org/juno/index.jsp?topic=/org.eclipse.platform.doc.user/concepts/concepts-25.htm. [Accessed: 24-Oct-2013].

[54] 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.

[55] G. Murphy, M. Kersten, and L. Findlater, “How are Java software developers using the Elipse IDE?,” *Software, IEEE*, 2006.

[56] NetBeans Team, “NetBeans IDE - The Smarter and Faster Way to Code,” 2013. [Online]. Available: https://netbeans.org/features/index.html. [Accessed: 25-Oct-2013].

[57] Oracle, “NetBeans IDE,” 2013. [Online]. Available: http://www.oracle.com/technetwork/developer-tools/netbeans/overview/index.html. [Accessed: 25-Oct-2013].

[58] NetBeans Team, “Base IDE,” 2013. [Online]. Available: https://netbeans.org/features/ide/. [Accessed: 25-Oct-2013].

[59] Project Qt, “Qt Features Overview,” 2013. [Online]. Available: http://qt-project.org/doc/qt-4.8/qt-overview.html. [Accessed: 25-Oct-2013].

[60] Project Qt, “Development Tools,” 2013. [Online]. Available: http://qt-project.org/doc/qt-5/topics-app-development.html. [Accessed: 25-Oct-2013].

Anexo A

Ferramentas para o Desenvolvimento de Software

Seguidamente serão apresentadas algumas *frameworks* que, sendo elas próprias *frameworks*, permitem a criação de *frameworks*. Primeiramente será apresentado o Visual Studio, seguindo-se o Eclipse, o Netbeans e o Qt.

Visual Studio

O Visual Studio é um conjunto bastante abrangente de ferramentas e serviços para o desenvolvimento de programas/aplicações, que têm como foco desktops, plataformas web, dispositivos móveis e a *cloud* [46]. Assim como a maioria das *frameworks*, este oferece ferramentas que agilizam os processos de *design*, escrita de código, *debug*, otimização e teste [47], no desenvolvimento de aplicações para plataformas Windows. A par destas funcionalidades, oferece ainda um sistema flexível que permite a colaboração de diferentes membros da mesma equipa, a trabalhar em plataformas diferentes, como o Eclipse e o XCode [48].

O ambiente Visual Studio apresentada na Figura 70, pode de facto ser visto como muito mais que uma plataforma de desenvolvimento, uma vez que oferece um conjunto de soluções que vão muito além da simples criação, geração e *debugging* de código, pois possibilita a gestão por completo da equipa e do trabalho desenvolvido ao longo do projeto [47].

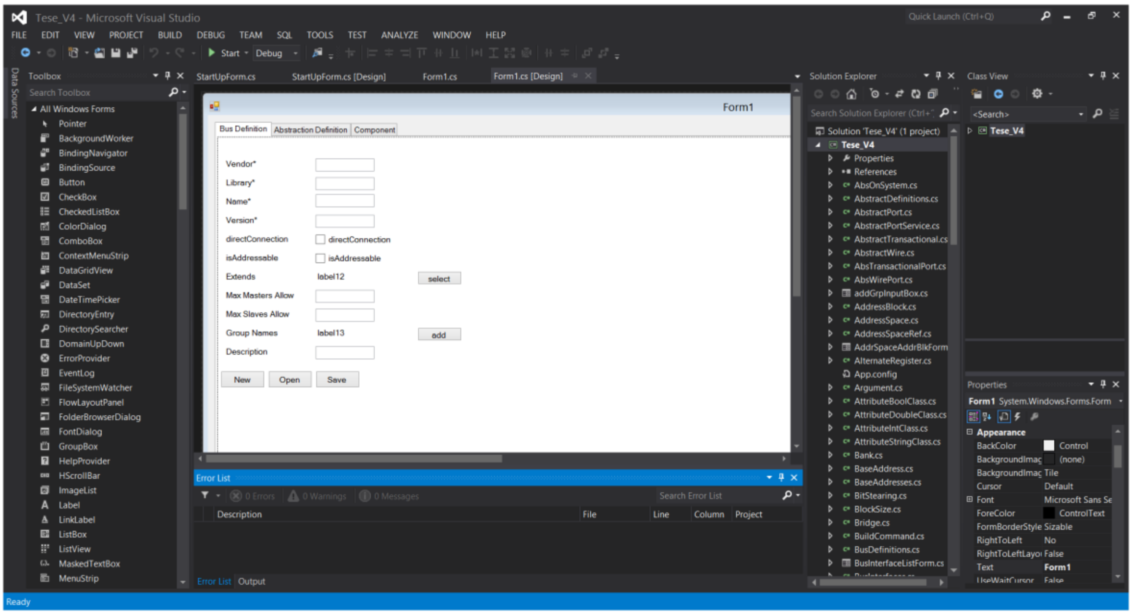


Figura 70 - Visual Studio 2012

As características acima citadas eram oferecidas através do serviço Team Foundation, disponível no Visual Studio 2012. No entanto, com o lançamento do Visual Studio 2013, a Microsoft lançou o Visual Studio online, cuja interface é apresentada na Figura 71, que substitui o anterior serviço Team Foundation e que, de acordo com o a Microsoft [33], possui as seguintes características:

- Número ilimitado de projetos e repositórios privados para o código desenvolvido;
- Acompanhar a evolução das tarefas a desempenhar e do código criado;
- Integração com Visual Studio, Eclipse e XCode;
- Suporte para gestão de projetos seguindo uma metodologia Agile;
- Possibilidade de utilizar a infraestrutura da *cloud* da Microsoft para correr *builds*.

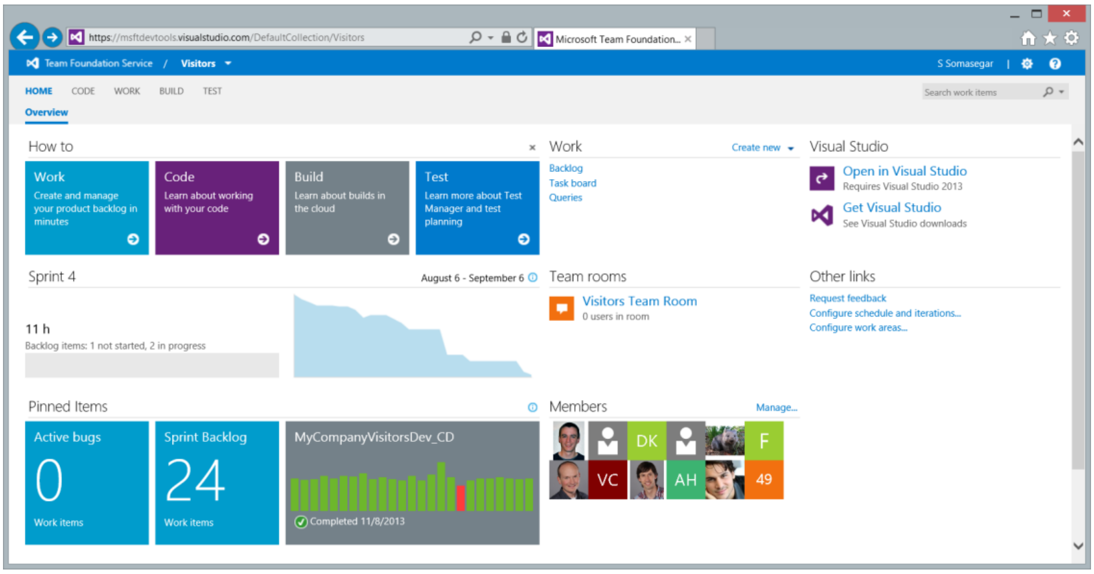


Figura 71 - Página inicial do Visual Studio Online [49]

Analisando apenas a parte relativa ao ambiente de desenvolvimento do Visual Studio, podemos dividir as suas características em dois grandes grupos: As capacidades essenciais e as ferramentas de Desenvolvimento & *Design* [47]. Dentro das capacidades essenciais oferecidas pela *framework* destacam-se as seguintes:

- Extensível: permite que sejam integradas com a *framework* ferramentas e SDKs desenvolvidos por qualquer programador;

- Personalizável: oferece um conjunto de configurações de sincronização e *layouts* para que o programador crie uma ambiente de desenvolvimento à sua medida;

- Pacote completo de ferramentas para *build* e otimização de aplicações;

- *Team Explorer*: oferece serviços de partilha e revisão de código entre elementos da mesma equipa;

- *Server Explorer*: garante a interação com serviços como Azure, servidores SQL e SharePoint.

No que diz respeito às ferramentas de Desenvolvimento & *Design*, o Visual Studio oferece suporte para o conjunto mais vasto de linguagens de programação (C++, C#, Javascript, HTML, CSS, VB.NET, XAML, SQL) e tirando partido de uma das suas características já mencionada, a *Team Explorer*, é possível obter um histórico do código desenvolvido [47]. Associando a esta ferramenta a integração do Lync (plataforma de comunicação da Microsoft unificada para empresas [50]) o trabalho em equipa é facilitado. Outras funcionalidades importantes oferecidas por esta *framework* são a identificação de código repetido e a possibilidade de *refactoring* do código [47].

Com o lançamento do Visual Studio 2012, foi acrescentada uma ferramenta com o nome de Blend. Através da utilização do Blend é possível editar a aparência de uma aplicação em *runtime*. Esta funcionalidade facilita o processo de *design* estético da aplicação, permitindo criar aplicações visualmente mais agradáveis para o utilizador.

Recentemente foi lançado o novo Visual Studio 2013, que para além de manter todas as funcionalidades oferecidas pelas versões anteriores, aumenta o leque de plataformas às quais dá suporte, entre elas [47]:

- Aplicações Desktop com sistema operativo Windows;
- Aplicações para a *Store* da Windows;
- Aplicações embebidas para plataformas da família Windows;
- Aplicações para WindowsPhone 8;
- Websites e Serviços;
- Windows Azure.

O Visual Studio permite ainda a criação de diagramas UML, mapas de dependências e mapas de código que facilitam a compreensão e o desenvolvimento de projetos complexos. Um exemplo de uma *framework* desenvolvida recorrendo ao Visual Studio é o CGI Studio da Fujitsu. Todas estas características contribuem para tornar a *framework* da Microsoft na plataforma mais utilizada em todo o mundo.

Eclipse

O projeto Eclipse foi desenvolvido pela IBM em 2001 e em 2004 foi criada a Eclipse Foundation, organização sem fins lucrativos, que tem como objetivo gerir o desenvolvimento contínuo do projeto Eclipse, dando apoio em quatro áreas que integram (i) *Information Tecnology infrastructure*, (ii) *Intelectual Property Managment*, (iii) *Development Communitie Support* e (iv) *Ecosystem Development* [51], [52].

Após a decisão de tornar o projeto Eclipse num projeto *open source*, a Eclipse Foundation recrutou inúmeros distribuidores de ferramentas de *software* incluindo a Borland, RedHat, Merant, SuSE e QNX. Desde então, juntaram-se ao projeto outras companhias como a HP e a Fujitsu.

O Eclipse é uma *framework* desenvolvida em java que oferece um conjunto de serviços para o desenvolvimento de projetos e ambientes de desenvolvimento a partir de um conjunto de *plug-ins*. Apesar de ser principalmente utilizado para desenvolver projetos em linguagem java, o Eclipse fornece a possibilidade de ser estendido devido à sua natureza *pure plug-in*, isto é, tudo no Eclipse é um *plug-in*. De facto, o Eclipse não é mais que um conjunto de subsistemas, constituídos por um ou mais *plug-ins*, construídos sobre um ambiente de execução (*runtime environment*). Esta caraterística, evidenciada na Figura 72, permite que todos os utilizadores tenham a possibilidade de adicionar funcionalidades ao Eclipse através, tornando-o assim numa *framework* flexível e à medida das necessidades de cada um [52].

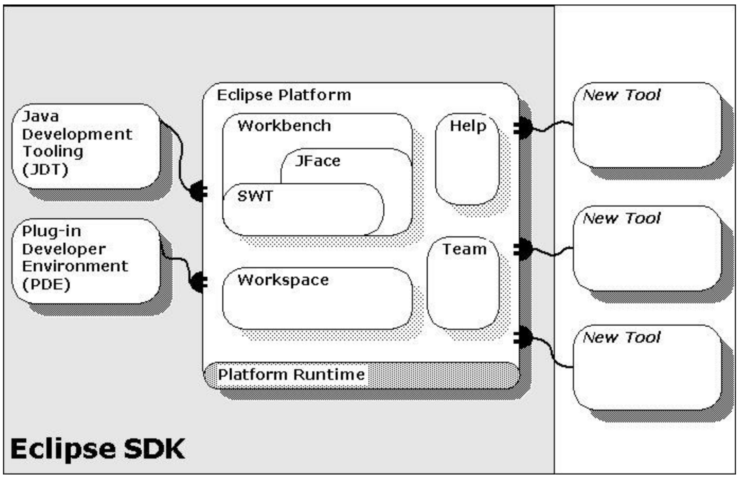


Figura 72 - Arquitetura da *framework* Eclipse [53]

Para além da plataforma base, o SDK do Eclipse incorpora duas ferramentas de grande importância, (i) o JDT (*Java Development Tools*) e (ii) o PDE (*Plug-in Development Environment*). O primeiro permite implementar um ambiente com todas as funcionalidades necessárias para o desenvolvimento em java, enquanto o segundo adiciona um conjunto de ferramentas que simplificam o desenvolvimento de *plug-ins* [53].

A *framework* Eclipse não se limita a oferecer um suporte para a programação em java. Apesar de nativamente só oferecer suporte ao java, através da instalação de *plug-ins* pode-se utilizar o Eclipse para desenvolver em linguagens como C/C++ e COBOL.

Os componentes genéricos do eclipse são o *Workspace*, responsável por gerir os projetos e ficheiros do utilizador, o *SWT* e *Jface*, que fazem a gestão do interface entre o Eclipse e o utilizador, o *Workbench*, que funciona como suporte ao interface gráfico, o *Team*, que interpreta o *Workspace* para efetuar controlo de versões, o *Debug* para verificação e validação das aplicações criadas, o *Help* e o *Update*, para dar suporte ao utilizador [54].

O desenvolvimento multiplataforma é assegurado aquando a compilação da aplicação, através da criação de um código intermédio que depois é interpretado e executado por uma JVM (*Java Virtual Machine*). À semelhança do Visual Studio, o Eclipse também permite o *refactoring* de código.

O facto de o Eclipse apresentar uma arquitetura baseada em *plugins* apresenta também algumas desvantagens principalmente na fase inicial de desenvolvimento e sobretudo em utilizadores pouco experientes. Este facto deve-se ao processo de instalação que pode tornar-se um pouco confuso dependendo da quantidade de *plugins* necessários para o desenvolvimento de uma aplicação.

Em suma, podemos dizer que o eclipse é uma *framework* multiplataforma, multilinguagem e multifuncionalidade, que oferece múltiplos *plugins* para a criação de plataformas de desenvolvimento como [55]:

- Editores de código fonte;
- Janelas de classes com hierarquia;
- Janelas de funções e atributos de uma classe;
- Lista de tarefas;
- Lista de problemas e erros de compilação;
- Janelas de preferências de utilizador;
- Ferramenta para integração de novos compiladores e *builders*;
- Janelas de informação de *debug* com *breakpoints*.

Um exemplo de uma *framework* desenvolvida recorrendo ao eclipse é o WebSphere Studio Workbench da IBM [52].

NetBeans

O Netbeans é uma *framework open source* e multilinguagem, em que as aplicações são desenvolvidas recorrendo a módulos [54]. Tal como as plataformas de desenvolvimento descritas anteriormente, o NetBeans é extensível e permite a compilação de código.

De acordo com a equipa desenvolvedora do Netbeans [56] e a Oracle [57], este *software* compõe um ambiente integrado de desenvolvimento que permite, de forma fácil e rápida, a criação de aplicações java para desktop, dispositivos móveis, aplicações web e aplicações em HTML5 com JavaScript, CSS e HTML. O Netbeans permite também a programação em PHP e C/C++.

É caracterizado por ser uma ferramenta modular que permite o desenvolvimento de aplicações para um vasto número de tecnologias. As principais funcionalidades oferecidas são (i) um editor de texto multi-linguagem avançado, (ii) um debbuger e (iii) ferramentas para o controlo de versões e gestão de equipas [58]. Tal como o Eclipse, o Netbeans é livre e conta com uma comunidade com um número vasto de utilizadores por todo o mundo.

Este *framework* é bastante similar ao Eclipse, diferenciando-se em pequenos detalhes. Segundo Faria [54], o Netbeans apresenta uma interface mais amigável para os utilizadores mais inexperientes pois a sua distribuição é realizada como um *software* normal, sendo necessário apenas uma instalação para que todas as ferramentas necessárias para iniciar a programação fiquem corretamente configuradas.

A forma como o Netbeans apresenta os erros também é um pouco distinta da que o Eclipse utiliza. Enquanto no Eclipse é possível visualizar um marcador no editor indicando o local do erro e algumas sugestões para o solucionar, no Netbeans os erros aparecem listados na consola com um indicador da linha e do ficheiro onde foram detetados [54].

Outra diferença entre o Netbeans e o Eclipse é a forma como guardam as configurações. Enquanto o Eclipse usa um diretório fora do Workspace, o Netbeans tem um *filesystem* que controla as informações de configuração, através de um repositório que armazena todos os dados internos [54].

Qt

O Qt foi criado para possibilitar o desenvolvimento de aplicações e interfaces que pudessem depois ser executadas em diversas plataformas e sistemas operativos sem ser necessário alterar o código do projeto [59].

O ambiente integrado de desenvolvimento disponibilizada pelo Qt dá pelo nome de Qt Creator e oferece um conjunto de ferramentas que guiam o utilizador pelo ciclo de vida completo de um projeto, desde a criação do projeto até ao momento de disponibilização da aplicação para a plataforma alvo. Este acompanhamento traduz-se na utilização de *wizards* que permitem automatizar algumas tarefas como criar os ficheiros necessários e definir as especificações do projeto com base nas escolhas do programador [60].

Para além de automatizar algumas tarefas, o Qt Creator permite acelerar o desenvolvimento de código através da verificação da sintaxe, coloração da semântica da linguagem, completação de código, ações de refactoring, entre outros.

De acordo com as informações disponibilizadas pela equipa do Qt [60], integradas com o Qt Creator existem ouras ferramentas como (i) o Qt Designer, para o desenvolvimento de interfaces gráfico, (ii) o qmake, que permite a compilação do mesmo projeto para diferentes plataformas, (iii) o Qt Linguist para localizar aplicações e (iv) o Qt Assistant, para visualizar toda a documentação relacionada com o Qt.

Por fim, importa salientar a enorme quantidade de bibliotecas disponibilizadas com o Qt e que facilitam bastante o trabalho do programador, acelerando o processo de desenvolvimento das aplicações.

Anexo B

Diagramas de Sequência

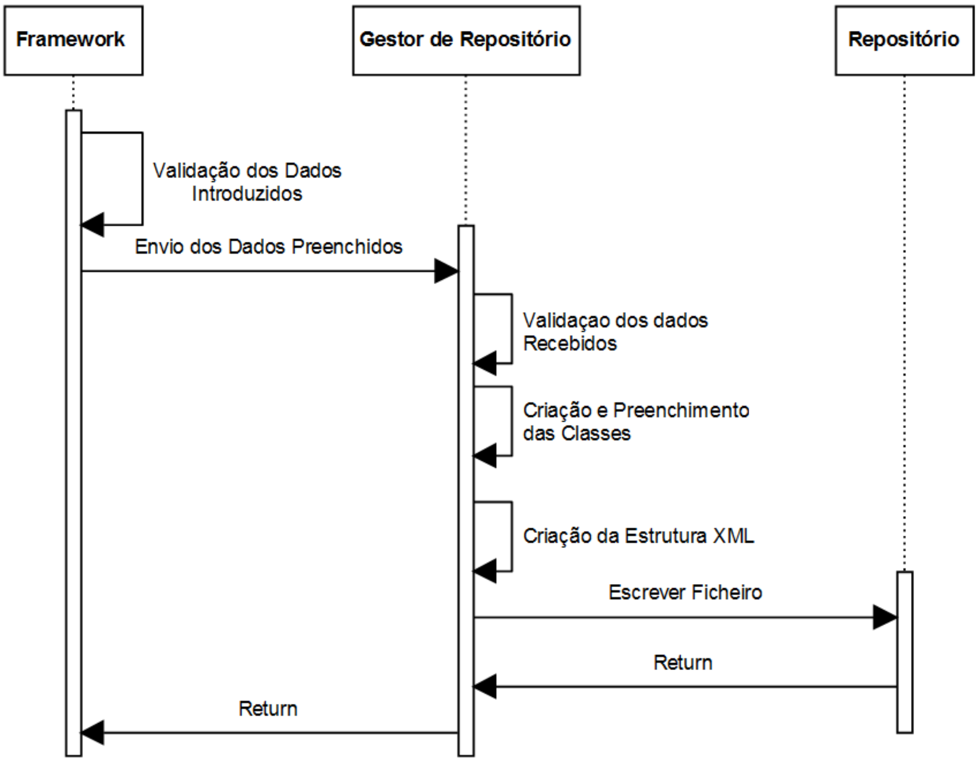


Figura 73 - Diagrama de Sequência de Escrita para o Repositório

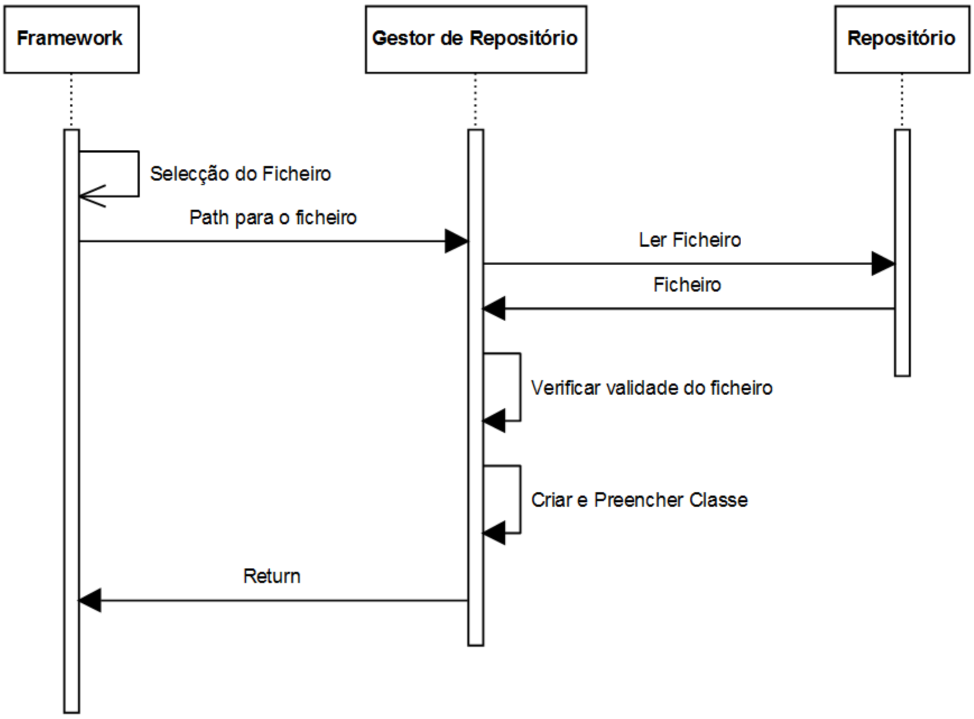


Figura 74 - Diagrama de Sequência para Leitura de IP do Repositório

Anexo C

Diagramas de Fluxo

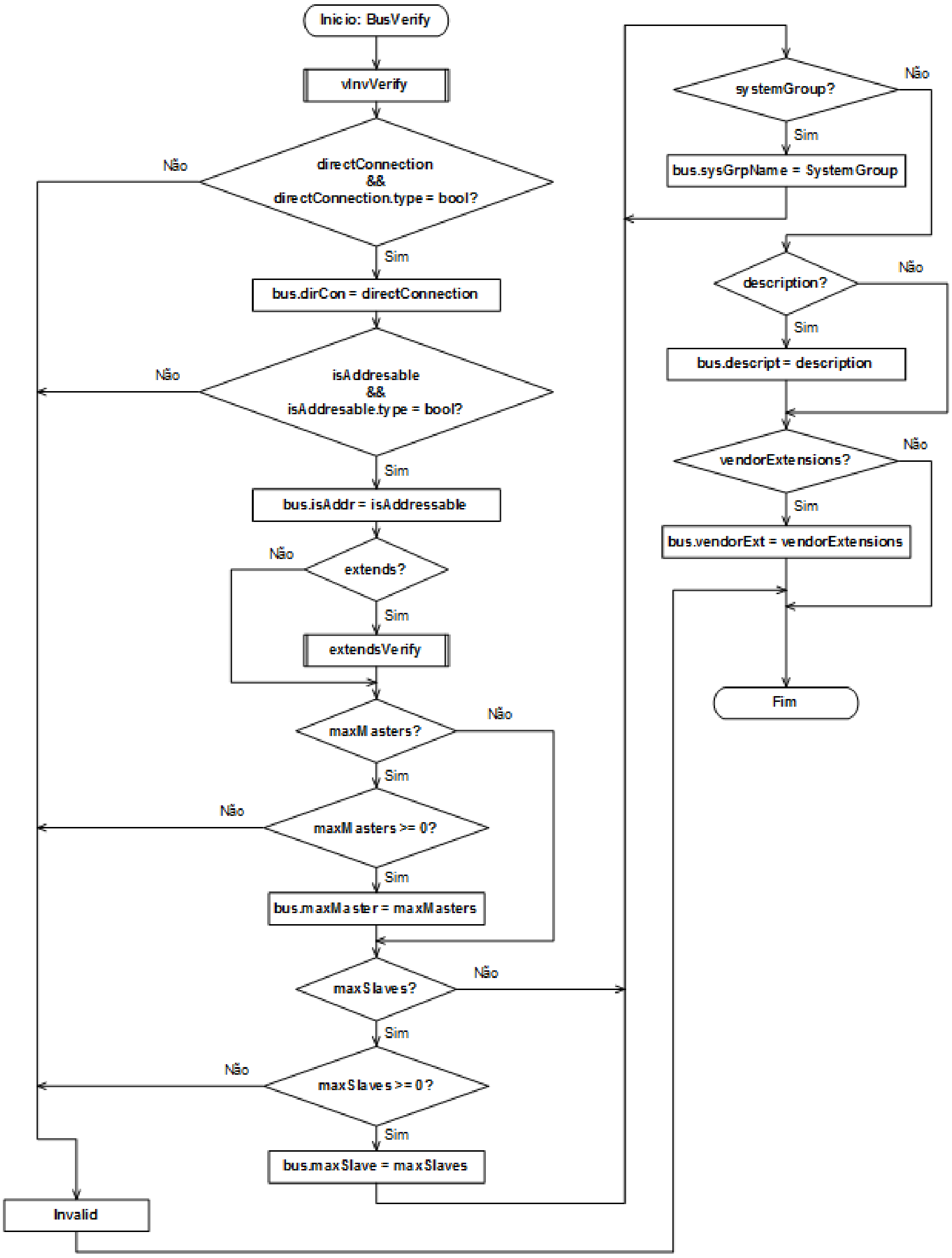


Figura 75 - Fluxograma para validação e IPs do tipo *Bus Definition*

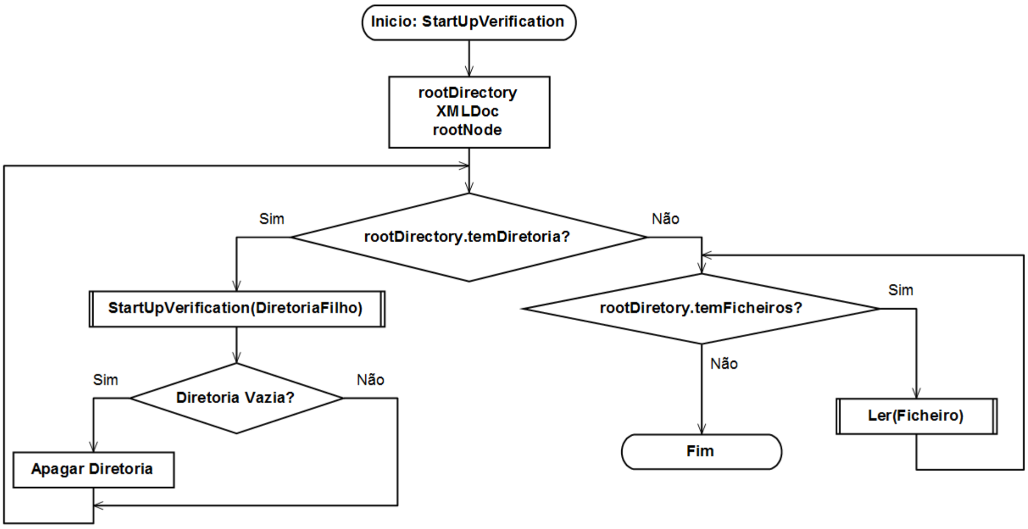


Figura 76 - Fluxograma de Verificação do Repositório

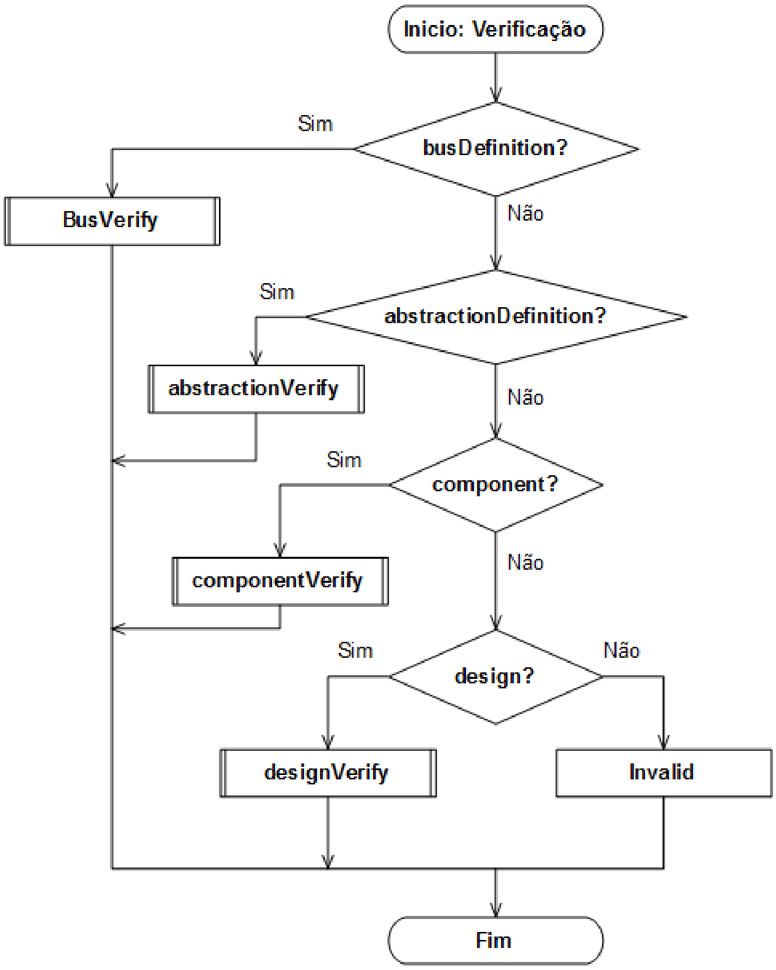


Figura 77 - Fluxograma geral para Verificação de IPs

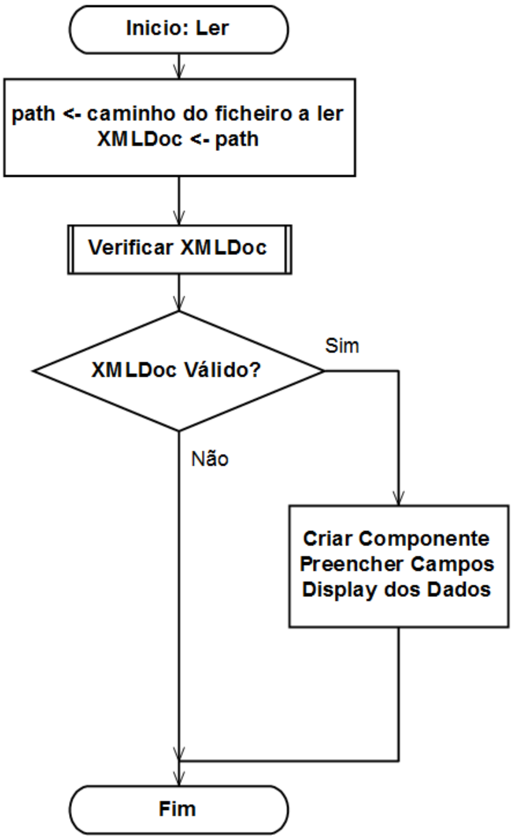


Figura 78 - Fluxograma para carregamento de IP

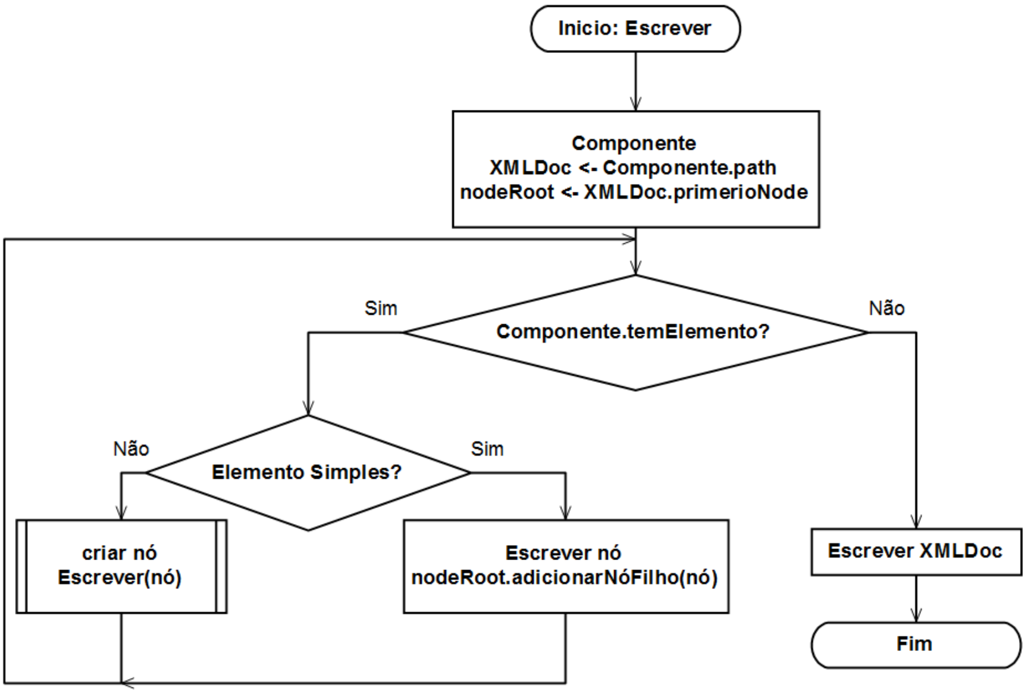


Figura 79 - Fluxograma para construção e escrita de novo IP

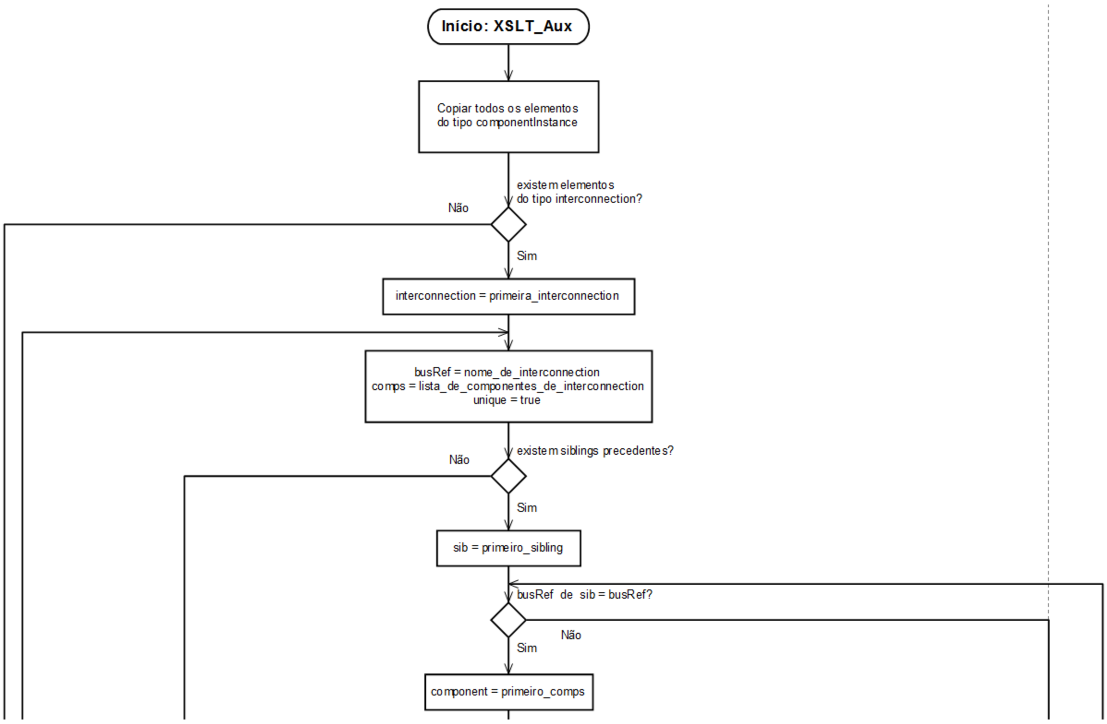


Figura 80 - Fluxograma para criação de XML auxiliar (Parte I)

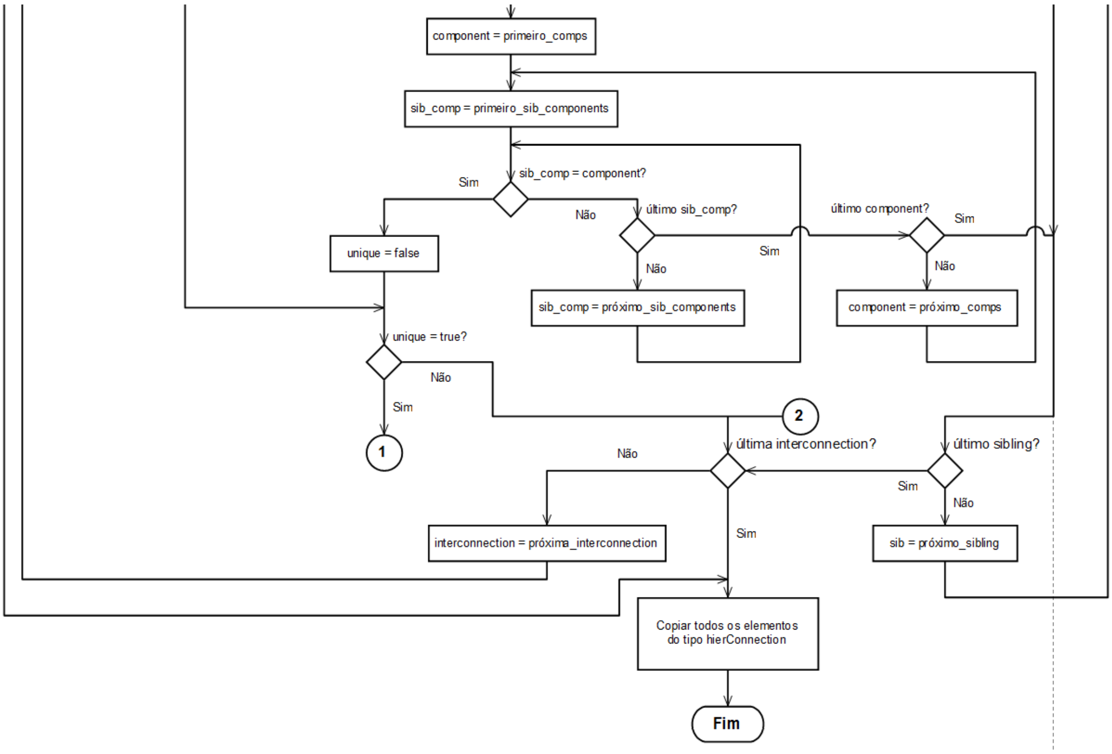


Figura 81 - Fluxograma para criação de XML auxiliar (Parte II)

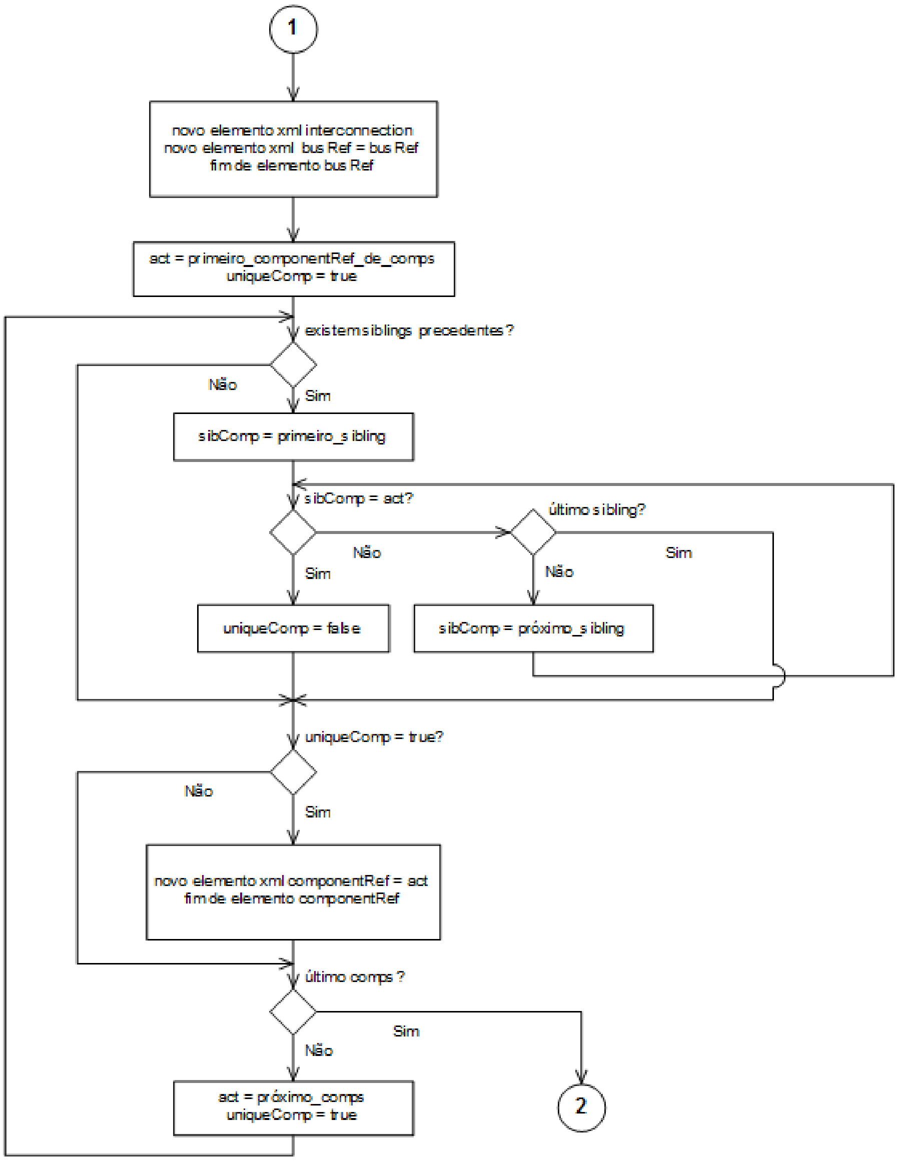


Figura 82 - Fluxograma para criação de XML auxiliar (Parte III)

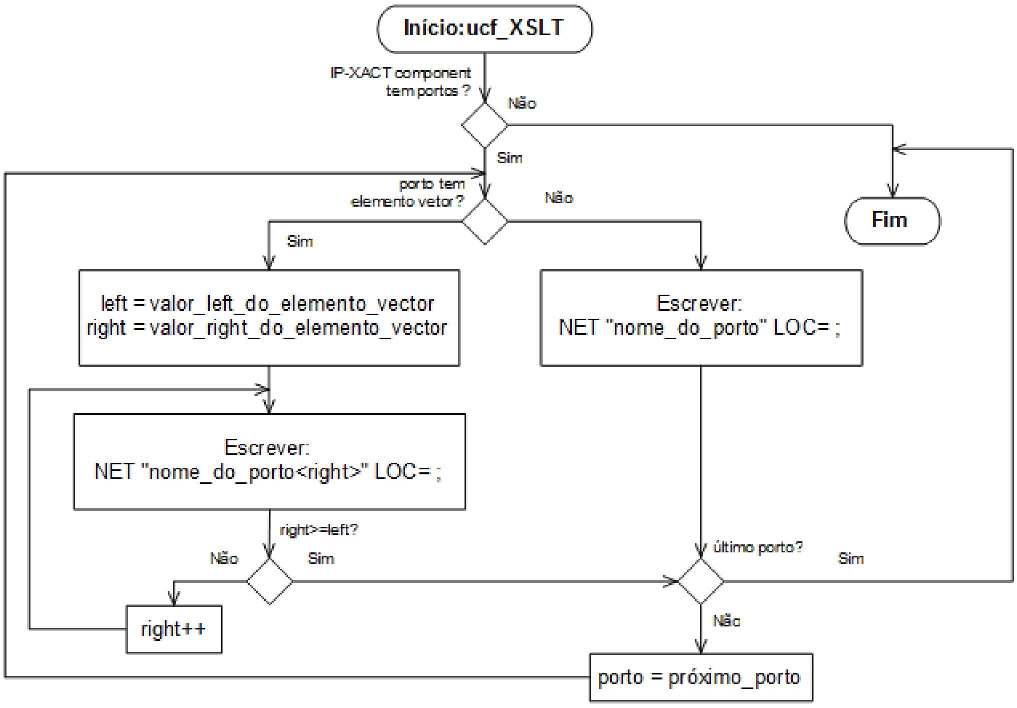


Figura 83 - Fluxograma para criação de ficheiro UCF

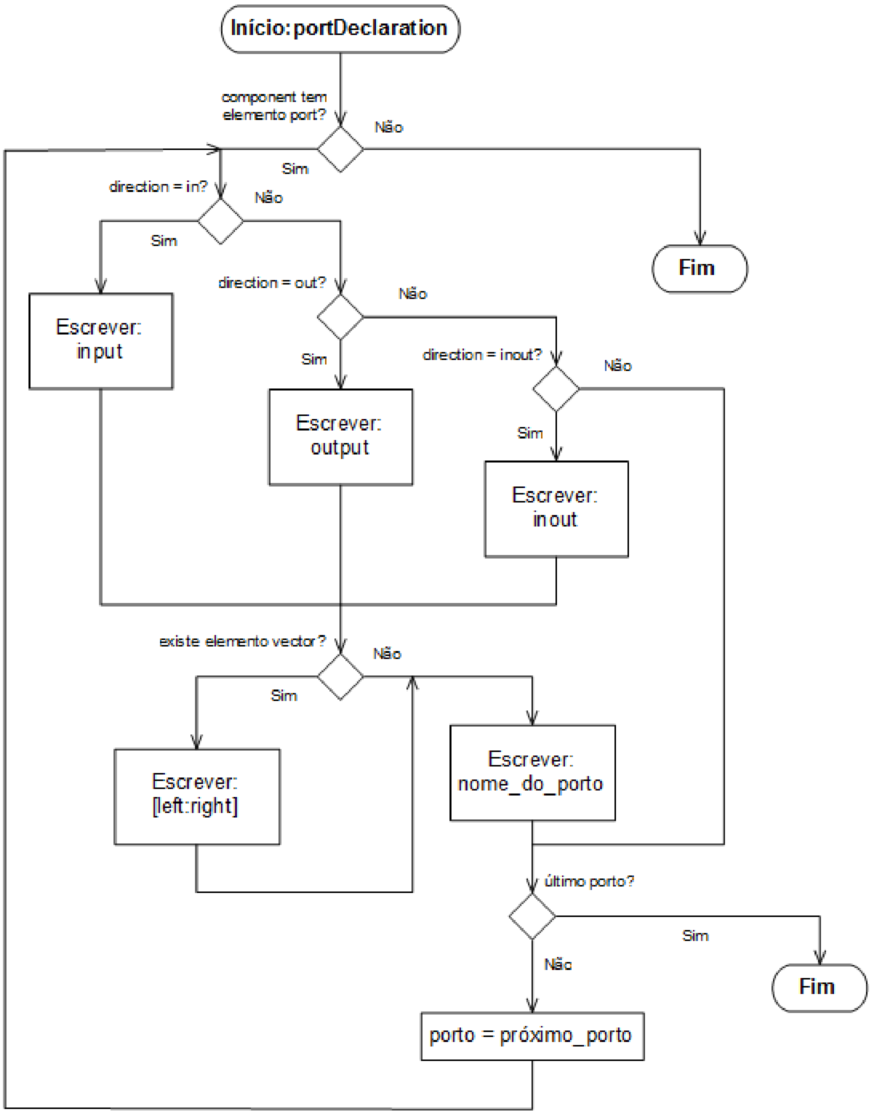


Figura 84 - Fluxograma para o processo de instanciação das interfaces do IP em Verilog

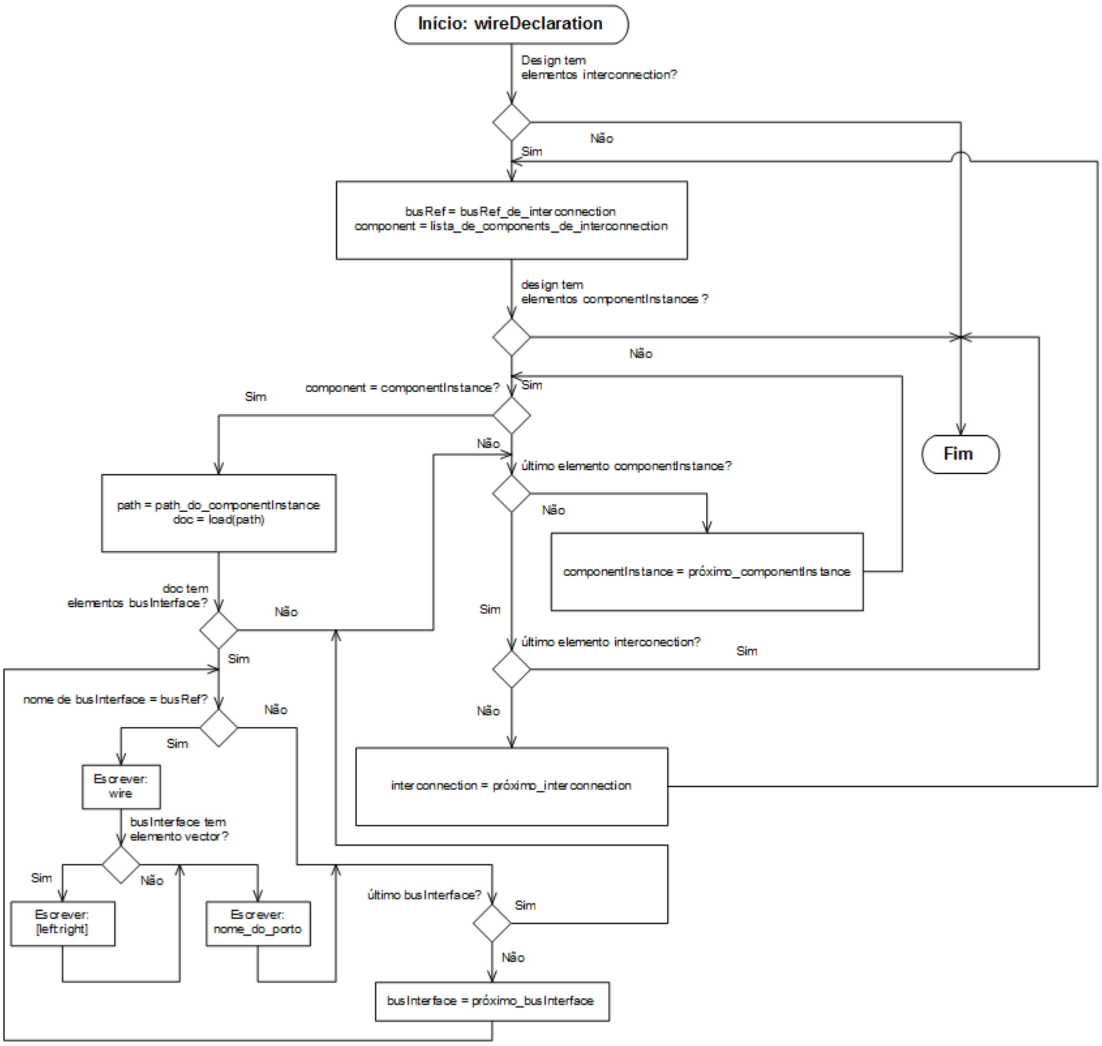


Figura 85 - Fluxograma para a declaração de interfaces *wire* em Verilog

Anexo D

Diagrama de Atividades

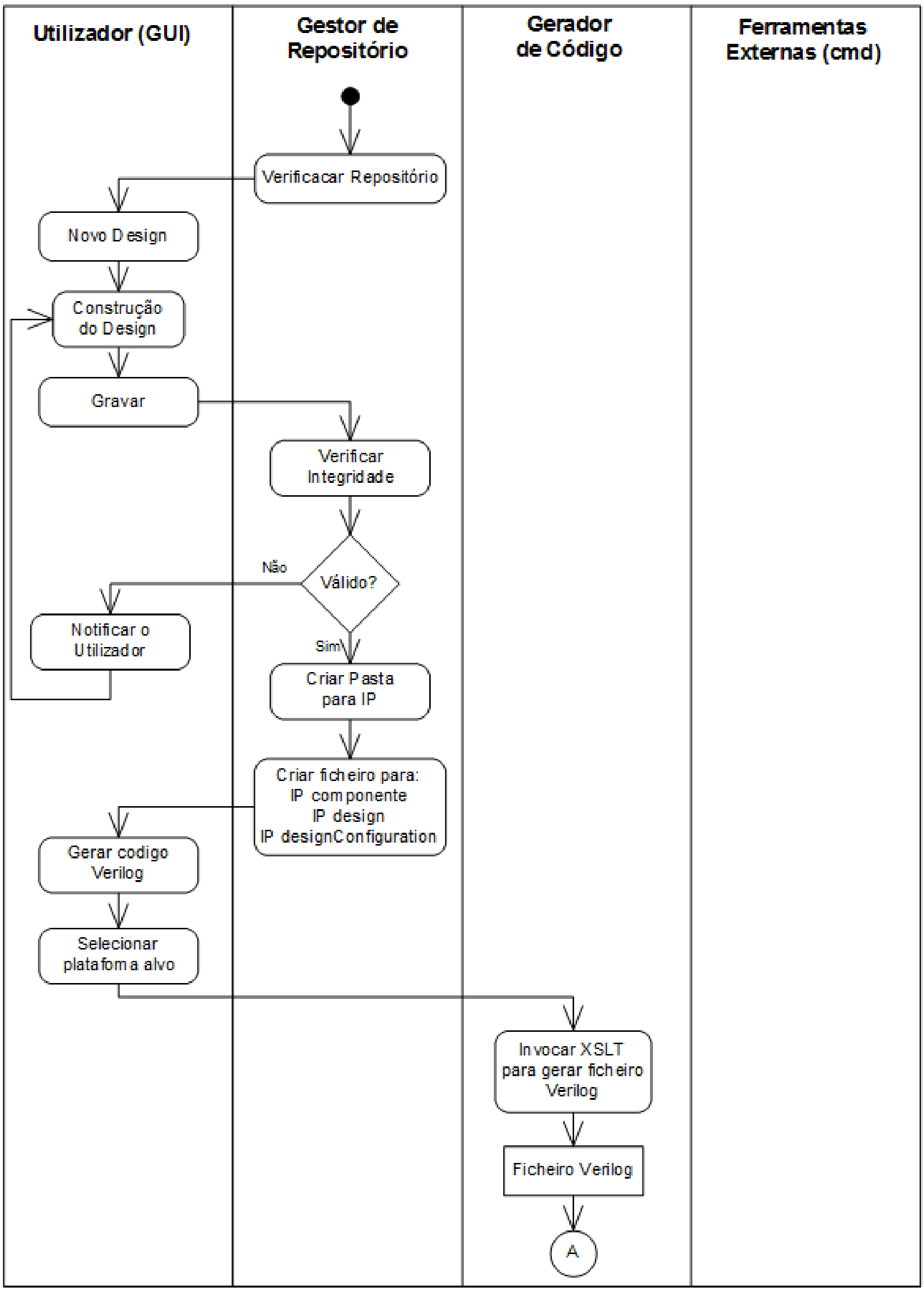


Figura 86 - Diagrama de Atividades para criação de Design (Parte I)

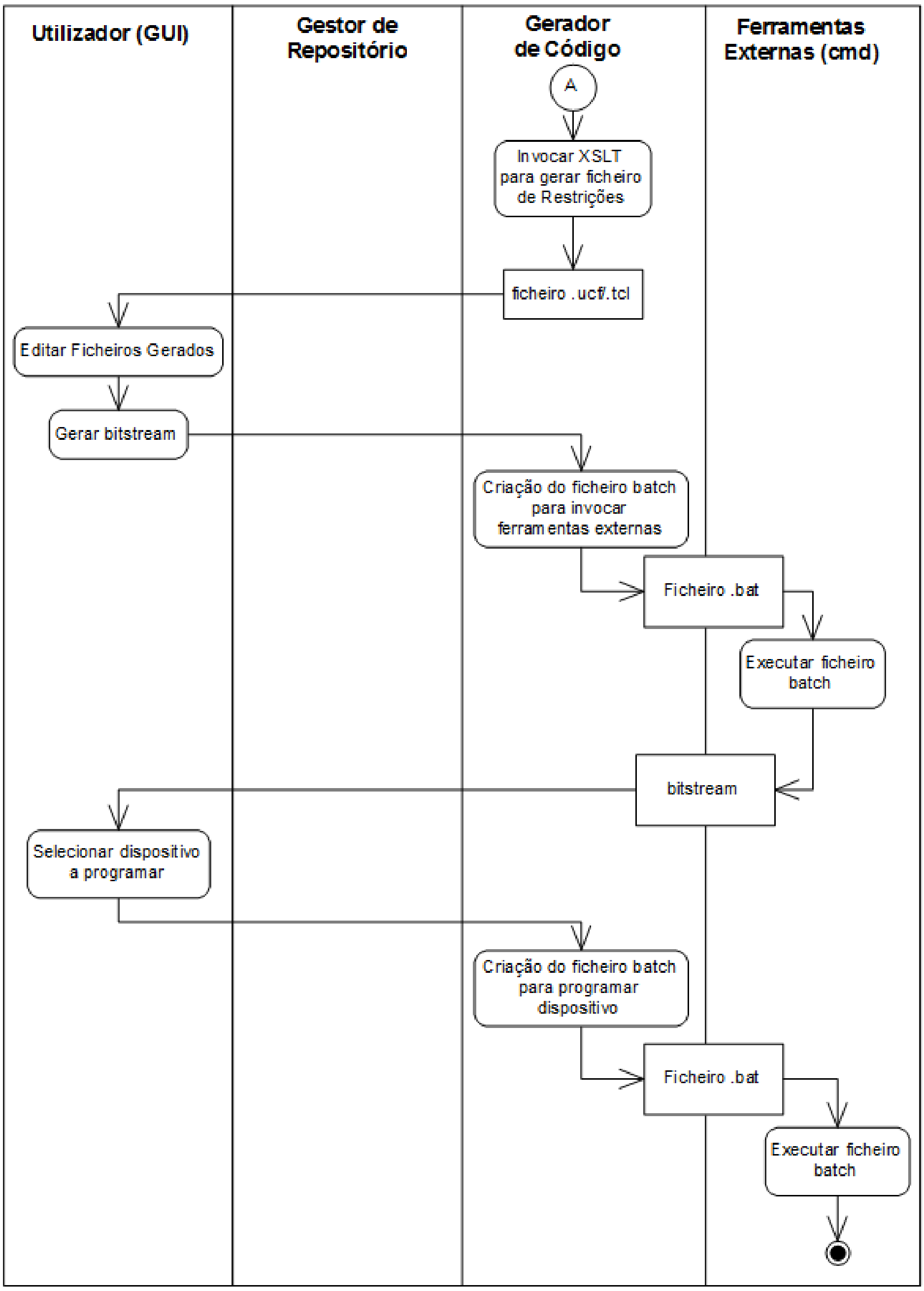


Figura 87 - Diagrama de Atividades para criação de Design (Parte II)

Anexo E

Plataformas de Teste

De maneira a testar o correto funcionamento do módulo referente ao gerador de código serão utilizadas duas plataformas, uma da Xilinx e outra da Altera. A plataforma da Xilinx é a placa de desenvolvimento Basys2, enquanto a plataforma utilizada no caso da Altera é a placa de desenvolvimento Cyclone II DE2-70. Seguidamente apresentam-se duas secções para descrever as especificações de cada um destes dispositivos.

Basys2

A plataforma de desenvolvimento Basys2 é uma placa desenvolvida em torno da FPGA da Xilinx Spartan3E e o controlador USB da Atmel AT90USB2 [36]. A placa vem equipada com um conjunto de circuitos de suporte, permitindo a criação de um conjunto de sistemas sem a necessidade de *hardware* auxiliar. A plataforma de desenvolvimento conta ainda com uma memória Flash ROM para guardar as configurações da FPGA, oito LEDs, quatro displays de sete segmentos, quatro botões e oito interruptores, um porto PS/2, um porto VGA de 8bits, um relógio configurável para frequências de 25/50/100 MHz e quatro conectores de seis pinos protegidos contra descargas electroestáticas (ESD) e circuitos com baixa impedância (short-circuits) [36].

A Basys2 é compatível com todas as versões da ferramenta ISE da Xilinx e o cabo USB, oferece não só a interface de programação, mas também a energia necessária para alimentar a placa. Para além do cabo USB a placa pode ser alimentada através de uma fonte externa, ligando-a ao conector de bateria existente na placa. No caso de se utilizar uma fonte de alimentação externa a diferença de potencial a assegurar deve encontrar-se no intervalo entre 3.5V-5.5V [36].

Para proceder à programação da FPGA pode utilizar-se a interface de programação ADEPT, disponibilizada gratuitamente pela Digilent. Este *software* deteta automaticamente a placa Basys2 permitindo a transferência de dados entre o PC e a placa [37]. Na Figura 88 é apresentada uma imagem da plataforma de desenvolvimento Basys2.

Figura 88 - Plataforma de Desenvolvimento Basys2

Cyclone II DE2-70

A plataforma de desenvolvimento Cyclone II vem equipada com uma FPGA Altera Cyclone II 2C70 e com o sistema USB Blaster para programar a FPGA e permitir ao utilizador controlar a aplicação. Esta interface suporta a programação da placa quer através de uma interface JTAG quer através de uma interface série ativa (AS). A FPGA Cyclone II 2C70 é constituída por 68416 LEs, 250 M4K blocos de RAM, 115200 bits de RAM, 150 multiplicadores embebidos, 4 PLLs e 622 pinos de I/O [38].

A placa vem ainda equipada com uma SSRAM de 2 Mbytes e duas SDRAM de 32 Mbytes cada, uma memoria Flash de 8 Mbytes, uma ranhura para cartões SD, quatro botões de pressão, dezoito interruptores, vinte e sete LEDs, um oscilador de 50 MHz e outro de 28.63 MHz que servem como fonte de relógio, um porto RS-232, um controlador Ethernet 10/100 com o respetivo conector, um conector VGA-out, um conector PS/2, um conector SMA e dois conectores de 40 pinos com proteção feita através díodos [38].

Esta plataforma de desenvolvimento é compatível com o *software* Quartus II, que deve ser utilizado para desenvolver projetos que tenham como plataforma alvo esta placa. Na Figura 89 é apresentada a placa descrita.

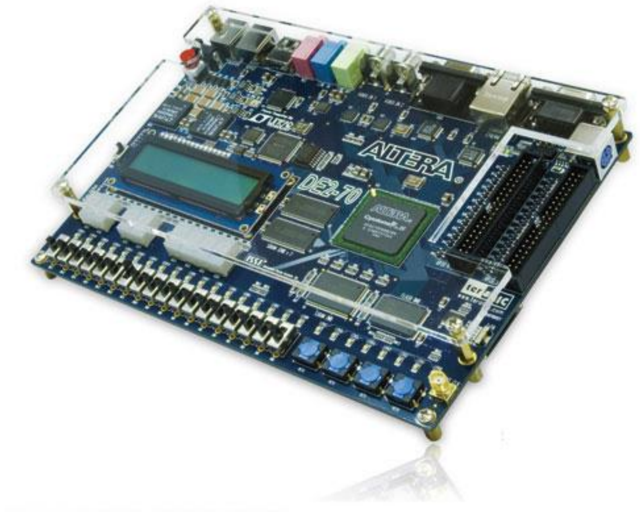


Figura 89 - Plataforma de Desenvolvimento Cyclone II DE2-70