UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA
ELÉTRICA E DE COMPUTAÇÃO

Autonomic Hardware Manager

Uma arquitetura de hardware autonômico usando a solução de repositório ativo de componentes

Julio Cesar Paulino De Melo

Orientador: Prof. Dr. Luiz Eduardo Cunha Leite

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e de Computação da UFRN (área de concentração: Engenharia de Computação) como parte dos requisitos para obtenção do título de Doutor em engenharia.

Número de ordem PPgEE: D140

Natal, RN, maio de 2015

Seção de Informação e Referência
Catalogação da Publicação na Fonte. UFRN / Biblioteca Central Zila Mamede

Melo, Julio Cesar Paulino de.
Autonomic hardware manager: uma arquitetura de hardware autonômico usando a solução de repositório ativo de componentes / Julio Cesar Paulino de Melo. – Natal, RN, 2015.
187f.

Orientador: Luiz Eduardo da Cunha Leite.

Tese (Doutorado em Engenharia Elétrica e de Computação) – Universidade Federal do Rio Grande do Norte. Centro de Tecnologia – Programa de Pós-Graduação em Engenharia Elétrica e de Computação.

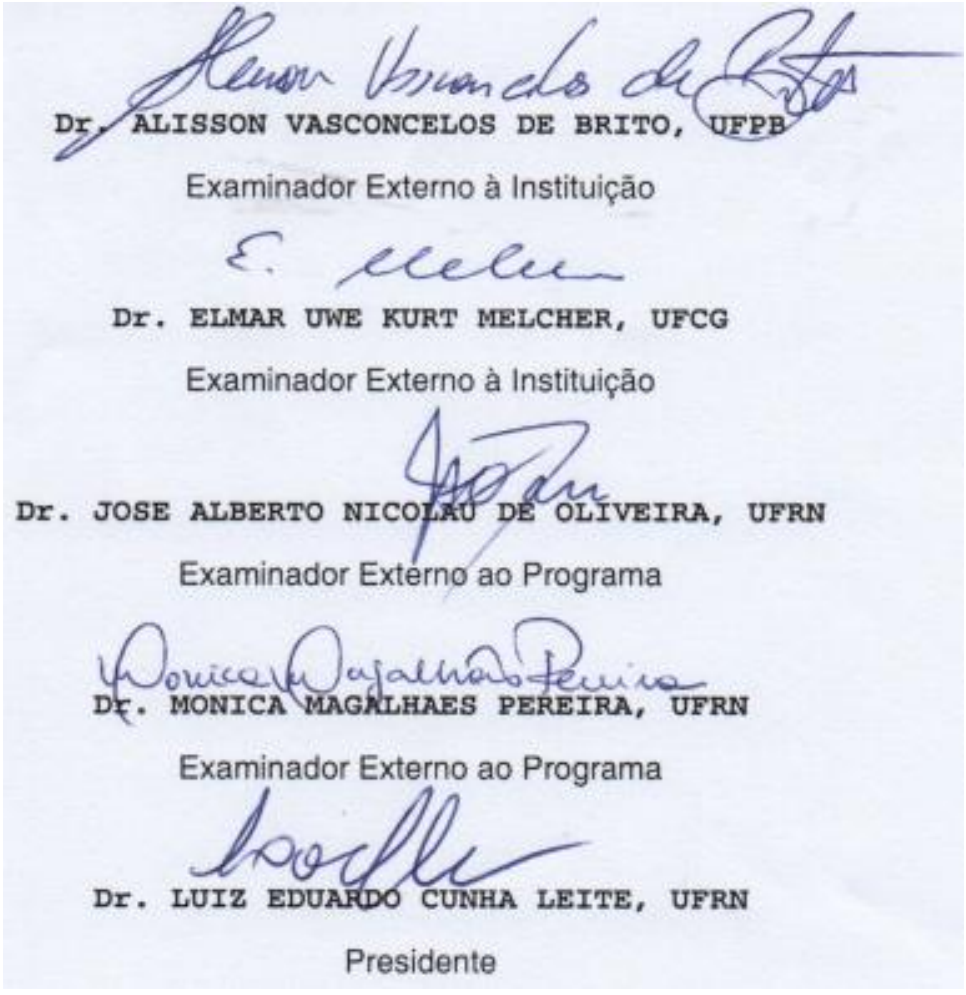
1. Sistemas embutido - Tese. 2. FPGA - Tese. 3. Reconfiguração dinâmica – Tese. I. Leite, Luiz Eduardo da Cunha. II. Título.

RN/UF/BCZM CDU 004.031.6

Autonomic Hardware Manager
Uma arquitetura de hardware autonômico usando a solução de repositório ativo de componentes

Julio Cesar Paulino De Melo

Tese de Doutorado aprovada em 29 de Maio de 2015 pela banca examinadora composta pelos seguintes membros:



Aos meus pais Francisco de Melo e Maria de Melo e meu irmão Samuel Melo, além de toda minha família pelo suporte e confiança durante todos esses anos...
A minha Namorada, Helliny, que deixou de estar comigo enquanto eu escrevia e trabalhava...

Dedico

AGRADECIMENTOS

Primeiramente, a Deus, pois a Ele todos devem algum agradecimento.

Ao Prof. Dr. Luiz Eduardo, pela dedicação nas correções e orientações neste período de aprendizado.

Ao Prof. Dr. Michael Hübner, que me deu suporte e me acolheu em seu laboratório na Alemanha.

A todos os meus outros amigos Professores, Prof. Dr. Luiz Marcos, Prof. Dr. Guido Lemos, Prof. Dr. Nicolau de Oliveira, Prof. Dr. Alisson Brito, Prof. Dr. Aquiles Burlamaqui, Prof. Dr. Rummenigge Dantas, Prof. Dr. Samuel de Azevedo, além de muitos outros que contribuíram diretamente com minha vida acadêmica e me guiaram até aqui.

A toda minha família, Tios, Tias e Primos que sempre me acolheram e estiveram sempre presentes.

Aos meus amigos do #Guardian por todo apoio, conversas e inspirações.

Aos meus amigos de Laboratório, Natalnet, Lar-ECT, ESIT-RUB por todos os trabalhos e cafés em conjunto.

Aos meus amigos da Dynavideo, mais competentes e inteligentes pessoas que já tive o prazer de conhecer e trabalhar em conjunto.

Aos meus outros amigos, que não fazem parte dos círculos supracitados, porém sempre me deram apoio em todos os nossos encontros e conversas.

A CAPES pelo apoio Financeiro.

RESUMO

AUTONOMIC HARDWARE MANAGER - UMA ARQUITETURA DE HARDWARE AUTONÔMICO USANDO A SOLUÇÃO DE REPOSITÓRIO ATIVO DE COMPONENTES

Esta Tese tem como objetivo desenvolver e implementar uma arquitetura para suporte a sistemas de Hardware Autonômicos, capaz de gerenciar o hardware em operação em dispositivos reconfiguráveis. A arquitetura proposta implementa mecanismos para manipulação, geração e comunicação de arquiteturas de hardware, usando a metodologia de Repositório Ativo orientado a Contexto. A solução consiste no desenvolvimento de uma arquitetura de Hardware-Software denominada *Autonomic Hardware Manager*, que contém um Repositório Ativo de Componentes de Hardware. Usando o repositório, a arquitetura se encarregará de gerenciar os sistemas embarcados conectados durante sua operação, possibilitando a implementação de características autonômicas como auto-gerenciamento, auto-otimização, auto-descrição e auto-configuração. A arquitetura proposta contempla também um metamodelo para representação do Contexto de Operação de sistemas de hardware. Esse metamodelo servirá de base para o desenvolvimento dos módulos de sensibilidade ao contexto, previstos na arquitetura do repositório ativo. Para fins de demonstração do funcionamento da arquitetura proposta, experimentos foram realizados com vistas a comprovar as hipóteses de pesquisa e alcançar cada objetivo desta tese. Três experimentos foram planejados e executados: o *Hardware Reconfigurable Filter*, que consiste em uma aplicação que implementa Filtro Digitais através de hardware reconfigurável; o *Autonomic Image Segmentation Filter*, que apresenta o projeto e implementação de uma aplicação autonômica de segmentação de processamento de imagens; por fim, o *Autonomic Auto Pilot* aplicação que consiste de um piloto automático para veículos aéreos não tripulados. Neste trabalho, a arquitetura das aplicações foi organizada em módulos, de acordo com as suas funcionalidades. Alguns destes módulos foram reimplementados em HDL e sintetizados em hardware. Outros módulos foram mantidos em software. Em seguida, a aplicações são integradas com o repositório AHM para possibilitar a sua adaptação aos diferentes contextos de operação, tornando-as autonômicas.

Palavras-Chave: Sistemas Embutidos, FPGA, Reconfiguração Dinâmica, Sistemas Autonômicos, Geração automática de Hardware.

ABSTRACT

AUTONOMIC HARDWARE MANAGER – AN AUTONOMIC HARDWARE ARCHITECTURE USING THE ACTIVE COMPONENT REPOSITORY APPROACH

This Thesis main objective is to implement a supporting architecture to Autonomic Hardware systems, capable of manage the hardware running in reconfigurable devices. The proposed architecture implements manipulation, generation and communication functionalities, using the Context Oriented Active Repository approach. The solution consists in a Hardware-Software based architecture called "Autonomic Hardware Manager (AHM)" that contains an Active Repository of Hardware Components. Using the repository the architecture will be able to manage the connected systems at run time allowing the implementation of autonomic features such as self-management, self-optimization, self-description and self-configuration. The proposed architecture also contains a meta-model that allows the representation of the Operating Context for hardware systems. This meta-model will be used as basis to the context sensing modules, that are needed in the Active Repository architecture. In order to demonstrate the proposed architecture functionalities, experiments were proposed and implemented in order to proof the Thesis hypothesis and achieved objectives. Three experiments were planned and implemented: the Hardware Reconfigurable Filter, that consists of an application that implements Digital Filters using reconfigurable hardware; the Autonomic Image Segmentation Filter, that shows the project and implementation of an image processing autonomic application; finally, the Autonomic Autopilot application that consist of an auto pilot to unmanned aerial vehicles. In this work, the applications architectures were organized in modules, according their functionalities. Some modules were implemented using HDL and synthesized in hardware. Other modules were implemented kept in software. After that, applications were integrated to the AHM to allow their adaptation to different Operating Context, making them autonomic.

Keywords: Embedded Systems, FPGA, Dynamic Reconfiguration, Autonomic Systems, Hardware Automatic Generation.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS

AHM	Autonomic Hardware Manager
API	Application Programing Interface
AS	Autonomic System
ASIC	Application Specifc Integrated Circuit
BRAM	Block Random Access Memory
CPS	Cyber Physical System
DSP	Digital Signal Processor
FPGA	Field Programing Gate Array
HDL	Hardware Description Language
LAR	Laboratório de Automação e Robótica
LS	Logic Slice
LUT	Lookup Table
OWL2.0	Web Ontology Language 2.0
RS	Register Slice
SBC	Software Baseado em Componentes
SRAM	Static Random Access Memory
UFRN	Universidade Federal do Rio Grande do Norte
UML	Unified Modeling Language
VANT	Veículo Aéreo Não tripulado
VHDL	VHSIC Hardware Description Language
XML	Extended Makeup Language

CAPÍTULO 1
INTRODUÇÃO

A utilização de hardware na solução de problemas computacionais é uma prática comum. Tanto no quesito desempenho como no quesito de economia de energia, a utilização de hardware em paralelo com software é cada vez mais difundida.

O desenvolvimento de circuitos integrados para aplicações específicas (*Aplication Specific Integrated Circuits* - ASICs) é cada vez mais necessário, porém fica cada dia mais caro devido à grande dinamicidade dos problemas envolvidos.

Problemas resolvidos com ASICs possuem, comumente, altas demandas de processamento que não podem ser satisfeitas através de uma solução puramente por software. Em contrapartida, os ASICs possuem um alto custo de desenvolvimento e implantação. Em cenários em que se deseja projetar um ASIC de baixo custo, a utilização de hardware reconfigurável promete uma solução que integre a facilidade de implantação do software com o poder de processamento de um hardware dedicado.

Com a utilização de hardware reconfigurável, é possível implementar um ASIC de forma que seja possível adaptá-lo para necessidades futuras. A tecnologia de reconfigurar hardware é bastante difundida. Contudo, com o crescimento da necessidade de flexibilidade dos hardwares dedicados, um novo nicho de dispositivos emergiu com a capacidade de serem alterados em tempo de execução (ALTERA CORPORATION, 2008) (DAVID, 2008).

Embora o tipo de reconfiguração varie de acordo com a Plataforma de Hardware utilizada, a essência da ideia é a mesma: permitir a modificação do hardware sem a necessidade de um desligamento geral do dispositivo. Essa característica proporciona principalmente economias significativas em área de circuito utilizada e potência consumida (PAPADIMITRIOU e DOLLAS, 2010).

As pesquisas relacionadas a esse tipo de hardware contemplam: ferramentas que possibilitam sua concepção e manipulação (LEE, YEN, *et al.*, 2010) (BRITO, SILVEIRA e MELCHER, 2010) (KOESTER, LUK, *et al.*, 2011); frameworks de design que levam em conta a presença desse tipo de hardware

(SOHANGHPURWALA, 2010); sistemas que fazem uso de reconfiguração dinâmica em aplicações específicas (PARK e BURLESON, 1998) (EL-ARABY, GONZALEZ e EL-GHAZAWI, 2009) (ANTONI, LEVEUGLE e FEHÉR, 2002).

Dentre os principais temas de pesquisa na área de reconfiguração dinâmica, essa Tese mais se relaciona com aqueles voltados à utilização e gerenciamento de hardware reconfigurável. Tais sistemas gerenciam o hardware que está ativo em uma dada plataforma de hardware, em um dado instante no tempo.

Embora existam sistemas capazes de gerenciar o hardware ativo em um dispositivo, pouco é mencionado a respeito da capacidade de geração de novas Arquiteturas de Hardware e do monitoramento dos Componentes presentes nas mesmas. Comumente, a implementação das interfaces para monitoramento dos Componentes, fica a critério da aplicação, caso as mesmas sejam necessárias.

Já no contexto da literatura referente ao desenvolvimento de software com arquitetura baseada em componentes (ou simplesmente Sistemas Baseados em Componentes - SBC), a capacidade de gerenciamento e monitoramento de componentes é uma característica muito mais consolidada.

A técnica de Desenvolvimento Baseado em Componentes consiste em dividir software em elementos menores denominados componentes, que são conectados a fim de implementar a funcionalidade do sistema. Devido à sua utilização em aplicações diversas, existem muitos estudos em direção à melhoria desse tipo de sistemas (COULSON, BLAIR, *et al.*, 2008) (BLAIR, COUPAYE e STEFANI, 2009) (YE, 2001) (MEROBASE CORPORATION).

Dentre as técnicas de gerenciamento e monitoramento de sistemas de software, surge o termo "Orientação a contexto" (DEY, 2001) ou "Sensibilidade a contexto" (SCHILIT, ADAMS e WANT, 1995). Um sistema orientado a contexto é capaz de modificar seus componentes durante sua operação, visando se adaptar a dadas modificações em parâmetros relevantes a seu funcionamento (BOLCHINI, CURINO, *et al.*, 2007).

Já no que diz respeito ao desenvolvimento de sistemas de hardware sensíveis ao contexto de operação, duas áreas que se destacam no âmbito das pesquisas mais recentes são os Sistemas Autonômicos (IBM, 2003) (HORN, 2001) e a os Sistemas Cyber-físicos (BAHETI e GILL, 2011) (LEE, 2008).

Sistemas Autonômicos (AS) foram concebidos a partir da observação de que a complexidade de determinadas aplicações crescia rapidamente, enquanto a sua capacidade de gerenciamento não aumentava na mesma velocidade. Diante desse cenário, foi necessária a criação de sistemas com capacidade de auto-gerenciamento, com base em regras de alto nível que relacionam variáveis importantes para o mesmo e ações de gerenciamento que devem ser tomadas quando certos eventos ocorrem.

Já os sistemas Cyber-Físicos(CPS) são descritos como "sistemas que apresentam forte interação com o ambiente que estão inseridos". Como nos sistemas autonômicos, esse tipo de sistema também possui características de auto-gerenciamento, dada sua relação com o ambiente onde está inserido. No entanto, uma característica relevante que permanece intrínseca aos CPSs é sua correlação direta com o tempo, devido a associação desse tipo de sistema com fenômenos físicos.

Tanto os ASs quanto os CPSs podem, de certa forma, ser classificados como sistemas orientados ao contexto. De fato, algumas pesquisas disponíveis na literatura (NAMI e SHARIFI, 2007) (STANISLAV e MICLEA, 2012) aplicam, no desenvolvimento destes tipos de sistema, técnicas e conceitos originalmente propostos para o desenvolvimento de sistemas de software orientados a contexto ou mesmo SBCs.

A modelagem de ASs, CPSs e até SBCs orientados a contexto é genérica o suficiente para permitir a sua aplicação no desenvolvimento de Sistemas de Hardware. No entanto, dentre esses três grandes grupos de sistemas, concentraremos o foco desta Tese na categoria de Sistemas Autonômicos que, como mencionado anteriormente, também exploram conceitos de SBSs orientados a contexto. Alguns trabalhos como (JOVANOVIC´, TANOUGAST e WEBER, 2008) (GUILLET, LAMOTTE, *et al.*, 2012) (GUILLET, LAMOTTE, *et al.*, 2012) (AN, RUTTEN, *et al.*, 2013) a serem detalhados no Capítulo 4, apresentam soluções que aplicam conceitos de sistemas autonômicos a sistemas de hardware, mostrando as vantagens da utilização das características comuns a tais sistemas nessa área.

Através da pesquisa bibliográfica, foi verificado que, dentre os trabalhos encontrados na literatura, os sistemas de gerenciamento propostos para sistemas de hardware autonômicos não exploram a geração de novas arquiteturas de hardware

durante a execução do sistema. O foco de tais trabalhos é direcionado para o chaveamento e/ou monitoramento dos componentes de hardware presentes no sistema em execução, sem a possibilidade de instalação de novos componentes.

Verificado esse aspecto, no escopo dessa Tese, é proposta uma arquitetura de gerenciamento de hardware autonômico que, aplicando conceitos de SBSs orientado a contexto, possibilitará a criação de sistemas autonômicos que poderão, durante a execução, explorar a geração, adaptação, atualização e implantação de arquiteturas de hardware.

A arquitetura desenvolvida será validada a partir da realização de experimentos, viabilizados através da implementação de duas aplicações de hardware autonômicas: o Autonomic Image Segmentation System e o Autonomic Auto-Pilot. Essas aplicações foram desenvolvidas de forma a possuir características de sistemas autonômicos, tais como: auto-reconfiguração, auto-gerenciamento e auto-atualização.

1.1 MOTIVAÇÃO

Em sistemas de hardware digital, é evidente a extensa utilização da componentização de cada entidade do sistema. Dessa forma, um sistema é composto pela interligação de diversos componentes, a fim de fornecer uma determinada funcionalidade. A prática de subdivisão de um sistema em componentes é também extensamente usada em ambientes de software, quando estes figuram entre os SBCs.

Adicionalmente, nos ASs, conceitos de SBCs orientados a contexto podem ser aplicados para viabilizar a implementação de alguns requisitos como auto-gerenciamento e auto-atualização.

Quadro 1.1 - Visão comparativa entre cinco características dos ASs, SBCs e sistemas de Hardware(HW). As células contendo "X" simbolizam que esse tipo de sistema contém aquela característica, células com "*" simbolizam que a característica pode ser adicionada dependendo da Aplicação, já o “-” representa que a característica em questão não está presente.

Característica/Tipo de Sistema	ASs	SBCs	Sistemas de HW
Modelagem Baseada em Componentes	*	X	X
Interação com elementos do ambiente onde inseridos	X	*	*
Integração de rotinas de auto-gerenciamento	X	*	-

Característica/Tipo de Sistema	ASs	SBCs	Sistemas de HW
Necessidade de modelagem com ferramentas de alto-nível	X	*	X
Modelagem tomando em consideração interconexão entre outros subsistemas	X	X	*

Fonte: Elaborada pelo Autor.

O Quadro 1.1 exibe cinco características comuns às classes de sistemas mencionadas (Sistemas Autonômicos, Sistemas de Software Baseados em Componentes e Sistemas puramente em Hardware). Nesta tabela, é possível observar semelhanças entre a classe dos sistemas puramente em hardware e as demais. Essas semelhanças são um indício da viabilidade de aplicação de técnicas de desenvolvimento de SBCs ao desenvolvimento de arquiteturas de hardware.

Como será mostrado em capítulos futuros, trabalhos recentes na área de Sistemas Autonômicos são focados no gerenciamento do hardware presente no dispositivo reconfigurável e não na geração automática de arquiteturas. Nos trabalhos estudados, a adição ou remoção de componentes são feitas durante a etapa de projeto, ou durante a execução, de forma manual.

Dessa forma, este trabalho é motivado pelo fato de que não foram encontrados na literatura, estudos nas áreas de gerenciamento de ASs ou de arquiteturas de hardware reconfigurável, no sentido da aplicação de técnicas de geração e atualização automática de Arquiteturas de Hardware.

Uma motivação mais específica ao desenvolvimento deste trabalho pode ser vista no projeto Veículo Aéreo não Tripulado (VANT), atualmente desenvolvido pelos laboratórios Natalnet e LAR da UFRN. Os sistemas integrados usados no controle e execução de missões de um VANT são sensíveis a vários fatores como: nível de combustível, ruído nos sensores, temperatura, localização da aeronave e parâmetros de controle utilizados. Esses atributos podem variar durante o período de operação e podem requerer sistemas de software e hardware que levem em conta essa dinâmica.

Tal adaptação poderia ser realizada por um sistema de hardware e software autonômico. Utilizando a arquitetura proposta nesta Tese, os componentes de hardware presentes no sistema embarcado podem ser monitorados, substituídos ou atualizados automaticamente durante a execução da missão planejada. O design de

um cenário de uso que pode ser aplicado por um VANT será mostrado com mais detalhes nas Seções 3.4 e 6.4.

1.2 DEFINIÇÃO DO PROBLEMA DE PESQUISA

Para definir melhor o problema de pesquisa é preciso ressaltar a definição de alguns termos que serão utilizados no texto que segue:

> ***Definição 1:*** *No escopo dessa tese o termo* ***Componente de Hardware*** *se refere à representação lógica de um hardware digital, seja ela feita em linguagem de descrição de hardware ou através de representação binária a mesma estando ou não sintetizada em um hardware reconfigurável.*
> ***Definição 2:*** *No escopo desta tese, chamamos* ***Arquitetura de Hardware*** *um conjunto de Componentes de Hardware interconectados estando a mesma sintetizada ou não em um hardware reconfigurável.*
> ***Definição 3: Contexto de Operação****, ou simplesmente* ***Contexto****, de um sistema de hardware é definido no escopo desta tese como um conjunto de variáveis dependentes da aplicação, que sejam relevantes para os requisitos funcionais ou não funcionais do sistema em operação.*

Com as definições desses termos, podemos observar que embora existam semelhanças entre sistemas de Hardware, SBCs e ASs, não foram encontrados trabalhos na literatura visando à aplicação de conceitos de SBCs a sistemas de hardware autonômicos, no que se diz respeito à geração de Arquiteturas de Hardware de forma automática. Desta forma, podemos definir o problema de pesquisa a ser abordado nesta Tese da seguinte forma:

> ***Problema de Pesquisa:*** *"Uma vez que as variáveis que influenciam no funcionamento e definem o contexto de um sistema de hardware são mapeadas, como podemos implementar um sistema autonômico que gere novas arquiteturas de hardware como reação a modificações ocorridas em tais variáveis?"*

1.3 METODOLOGIA

Uma estratégia que pode ser utilizada para solução do Problema de Pesquisa proposto consiste em projetar um arcabouço capaz de modificar a arquitetura de hardware em tempo de execução. Tal tarefa pode ser realizada através de uma API ou framework que possibilite a modificação do hardware presente em um dispositivo de hardware reconfigurável, a partir de observações feitas em seu estado atual.

Dada tal arquitetura, uma descrição de Variáveis de Contexto deve ser adotada, bem como regras que definem quando e como o sistema será modificado. Tal descrição pode utilizar uma ferramenta formal como uma ontologia definida através da *Ontology Web Language* 2.0 (OWL2.0) (L.MCGUINNESS e HARMELEN, 2004) ou através de um conjunto simples de variáveis e regras.

Essas variáveis definirão quais elementos do sistema devem ser observados ao longo do tempo e quais ações devem ser executadas caso um dado evento ocorra. A arquitetura geral pode suportar esse tipo de definição através de um sistema reativo, que irá identificar quando cada modificação é necessária e disparar uma modificação no sistema.

Quando uma modificação for necessária, a arquitetura de gerenciamento deve ser capaz de gerar uma nova Arquitetura de Hardware e incorporá-la à Aplicação em execução. A geração de Arquiteturas de Hardware pode ser feita sob demanda, em uma entidade separada, através do uso de ferramentas que a possibilitem.

O problema então será abordado com o desenvolvimento de duas entidades principais, que fazem parte de uma arquitetura maior denominada *Autonomic Hardware Manager* (AHM). Das duas entidades, uma será implementada na plataforma de hardware, composta de elementos de Hardware e Software que juntos compõem uma entidade chamada Cliente AHM, capaz de controlar sua configuração interna e requisitar novas arquiteturas. A segunda entidade, chamada Repositório AHM, será capaz de gerar novas arquiteturas quando requisitada, empregando a técnica de Repositório Ativo de Componentes.

Adicionalmente, será usada uma representação de contexto simplificada, usando uma tabela variável-valor junto com regras que serão guardadas no

Repositório AHM. O repositório usará um mecanismo de monitoramento para disparar a geração/requisição de novas Arquiteturas de Hardware.

Na implementação serão usadas plataformas de hardware reconfigurável FPGA. Dado que os algoritmos de gerenciamento de arquitetura presentes no Cliente AHM serão implementados em software, o dispositivo de hardware em questão precisa possibilitar o desenvolvimento e implantação de software. Para isso, podemos utilizar o desenvolvimento de hardware orientado à plataforma (DAYA, 2009).

A arquitetura final será testada através da implementação de três cenários de uso, o *Hardware Reconfigurable Filter*, o *Autonomic Image Segmentation System* e o *Autonomic Autopilot*. Após a implementação, testes para verificação das capacidades autonômicas das aplicações serão realizados com o objetivo de mostrar a capacidade do sistema proposto de gerar e atualizar arquiteturas de hardware de forma automática.

1.4 TESE

O objetivo principal desta Tese é projetar e implementar um arcabouço que permita o desenvolvimento de sistemas autonômicos de hardware, possibilitando a geração de Arquiteturas de Hardware que melhor se adaptem ao estado atual da aplicação. Portanto uma resposta hipotética ao problema a ser resolvido é:

> ***Hipótese Principal:*** *Uma vez que as variáveis que influenciam no funcionamento e definem o contexto de operação de um sistema de hardware são mapeadas, podemos usar a técnica de Repositório Ativo de Componentes Orientado ao Contexto, para implementar um mecanismo de gerenciamento para sistemas autonômicos que modele e adapte o sistema à dinâmica dessas variáveis.*

O mecanismo de gerenciamento deverá ser capaz não só de gerenciar e monitorar as variáveis de contexto e os componentes em execução na aplicação. Ele deverá ser capaz também de gerar novas arquiteturas de hardware para aplicação, quando for necessário e possível.

No entanto, para que a hipótese seja plausível, os pressupostos teóricos apresentados abaixo serão assumidos:

PS1 - *A Plataforma de Hardware, na qual a Aplicação Autonômica irá executar, deve permitir reconfiguração do hardware.*

PS2 - *É possível acessar as ferramentas de síntese de hardware, providas pelos fabricantes das plataformas de hardware reconfigurável, através de APIs de software.*

PS3 - *É possível implementar a comunicação com os componentes de hardware, desenvolvidos com as ferramentas do fabricante, através de software embarcado.*

Para comprovar a Hipótese Principal, dividimos a mesma em hipóteses secundarias que são enumeradas a seguir:

Hipótese sobre a representação de componentes de hardware:

HS1 - *É possível adaptar um modelo de componentes de software para que possa ser aplicado a componentes de hardware.*

Hipótese a respeito da modificação dinâmica do hardware:

HS2 - *O sistema de hardware pode ser reconfigurado, dentro do sistema embarcado, através de interfaces de software.*

Hipótese a respeito da geração dinâmica de Hardware:

HS3 - *É possível gerar novas arquiteturas de hardware para o sistema gerenciado, usando componentes disponíveis, um engenho de software e as ferramentas de síntese de hardware providas pelo fabricante.*

Hipótese a respeito do mecanismo de comunicação:

HS4 - *É possível desenvolver uma arquitetura de hardware embarcado em dispositivo reconfigurável na qual seja possível observar o estado de cada componente da arquitetura, independentemente da quantidade de componentes e interfaces presentes na mesma.*

Hipóteses sobre representação de contexto:

HS5 - *Uma representação das variáveis de contexto e seus possíveis valores pode ser construída visando fornecer ao sistema um meio de observa-las de forma automática.*

HS6 - *Regras de alto nível podem ser escritas através de relações condicionais entre as variáveis que descrevem o contexto e seus valores esperados, visando adaptar o sistema a possíveis condições refletidas em sua descrição de contexto.*

Uma vez provadas verdadeiras as hipóteses secundárias e respeitados os pressupostos teóricos, a tese defendida neste trabalho pode ser sumarizada como:

Tese: *"Aplicando a técnica de Repositório Ativo Orientado a Contexto, no desenvolvimento de um sistema de hardware implementado em um dispositivo reconfigurável, é possível criar uma arquitetura de hardware/software autonômica, capaz de reagir a alterações no seu contexto de operação, que por sua vez é modelado através de variáveis de contexto."*

1.5 OBJETIVO PRINCIPAL E ESPECÍFICOS

Como pôde ser observado na Hipótese Principal, o objetivo deste trabalho é desenvolver um mecanismo que possibilite o a implementação de Sistemas de Hardware Autonômico, através do emprego da técnica de Repositório Ativo de Componentes orientado ao Contexto. A fim de prover adaptação arquitetural desses sistemas à dinâmica das Variáveis de Contexto relevantes a seu funcionamento.

Para isso, a implementação de uma arquitetura de gerenciamento de componentes de hardware, a escolha e adaptação de um modelo de componentes, entre outros objetivos são necessários. A fim de comprovar as hipóteses secundárias, provando assim a Hipótese Principal, os seguintes Objetivos Específicos foram planejados:

O1 - *Escolher um Modelo de Componentes a ser usado para encapsular os componentes de hardware a serem gerenciados.*

O2 - *Projetar e implementar a arquitetura do Cliente AHM e Repositório AHM.*

O3 - *Projetar e implementar uma solução que possibilite a observação de elementos internos da arquitetura de hardware.*

O4 - *Projetar e implementar uma solução de modelagem de contexto para ser implementada nos Cliente AHM e Repositório AHM.*

O5 - *Projetar e implementar uma adaptação dos Cliente AHM e Repositório AHM para incorporar a modelagem de contexto.*

O6 - *Implementação de aplicações de teste para avaliar o desempenho da arquitetura.*

1.6 RESTRIÇÕES DA ARQUITETURA AHM

Neste Documento serão mostrados todos os elementos arquiteturais usados na implementação do AHM, porém, como enumerados nos pressupostos teóricos, algumas restrições existem na aplicação da solução apresentada. Essas restrições dizem respeito à Plataforma de Hardware escolhida para a Aplicação Embarcada e à Plataforma do Repositório escolhida para implementação da Aplicação do Repositório Ativo.

As restrições a respeito da Plataforma de Hardware são enumeradas a seguir:

- A Plataforma de Hardware precisa ser capaz de executar software e que possa se comunicar com o hardware configurado na região reconfigurável.
- A Plataforma de Hardware precisa ser capaz de armazenar os *bitstreams* das Arquiteturas de Hardware recebidas.
- A Plataforma de Hardware precisa ser fornecer um meio de comunicação com o Repositório AHM.

Além destas a Plataforma do Repositório, precisa prover os seguintes requisitos:

- A Plataforma precisa de um meio para acessar as Ferramentas de Síntese do fabricante através de APIs de software, linguagem de script ou executáveis externos.
- A Plataforma do Repositório precisa fornecer um meio de comunicação compatível com a Plataforma de Hardware.

Portanto, neste trabalho, serão abordadas Aplicações Embarcadas do Tipo Orientadas à Host que constituem um Sistema Distribuído através da comunicação, usada para gerenciamento, entre a Aplicação Embarcada e o Repositório.

Aplicações que não se encaixem nessa categoria, não serão suportadas pela arquitetura proposta nesta Tese.

Um tutorial mais detalhado enumerando os passos para implementar um dos sistemas apresentados como exemplo nesta tese, que melhor representa as restrições enumeradas, encontra-se no APÊNDICE A.

Outra restrição, que pode ser visualizada ao observarmos os resultados que serão apresentados no Capítulo 6, diz respeito à aplicação da geração Autonômica de Hardware em aplicações de Tempo Real.

Devido à utilização das ferramentas de processamento de HDL o tempo para gerar uma Arquitetura de Hardware, durante a execução do sistema, é relativamente alto. Portanto não é possível, até o momento, usando a metodologia apresentada nesta Tese, gerar e reconfigurar Arquiteturas de Hardware em Tempo Real, sendo essa uma das principais limitações da arquitetura AHM.

Embora não seja possível gerar arquiteturas de hardware em tempo real, o AHM possibilita APIs para reconfiguração de tais arquiteturas em tempo hábil. De forma que a reconfiguração dinâmica de Arquiteturas de Hardware pode ser aplicada em sistemas de tempo real, uma vez que as arquiteturas estivessem pré-geradas e armazenadas na Plataforma de Hardware.

1.7 ORGANIZAÇÃO DO TRABALHO

Esse trabalho está dividido em sete Capítulos. O Capítulo 2 apresenta um pouco da teoria necessária ao entendimento dos conceitos que serão usados no decorrer do texto. O Capítulo 3 apresenta um conjunto de possíveis aplicações nas quais a arquitetura proposta nesta Tese pode ser aplicada. O Capítulo 4 agrupa uma coletânea de trabalhos que julgamos mais relacionados, bem como comparações entre eles e a arquitetura apresentada nesta Tese. O Capítulo 5 apresenta a arquitetura geral proposta nesta Tese. O Capítulo 6 apresenta os experimentos e análises realizados, a fim de avaliar a arquitetura proposta, mediante comprovação das hipóteses de pesquisa e consecução dos objetivos propostos na Tese. Por fim, o Capítulo 7 apresenta as conclusões, considerações finais e trabalhos futuros.

CAPÍTULO 2
EMBASAMENTO TEÓRICO

Para melhor entendimento dos objetos desenvolvidos e apresentados nesta Tese alguns tópicos de embasamento precisam ser elucidados. Nesta seção serão discutidos termos e conceitos que serão usados, no decorrer da Tese, como base para o desenvolvimento dos experimentos e da arquitetura proposta.

Inicialmente será tratada a tecnologia FPGA e seus aspectos mais importantes. Após isso, abordaremos os sistemas de software orientados a componentes. Por fim, abordaremos o desenvolvimento de sistemas de software orientados à componentes com o uso de Repositórios.

2.1 TECNOLOGIA FPGA

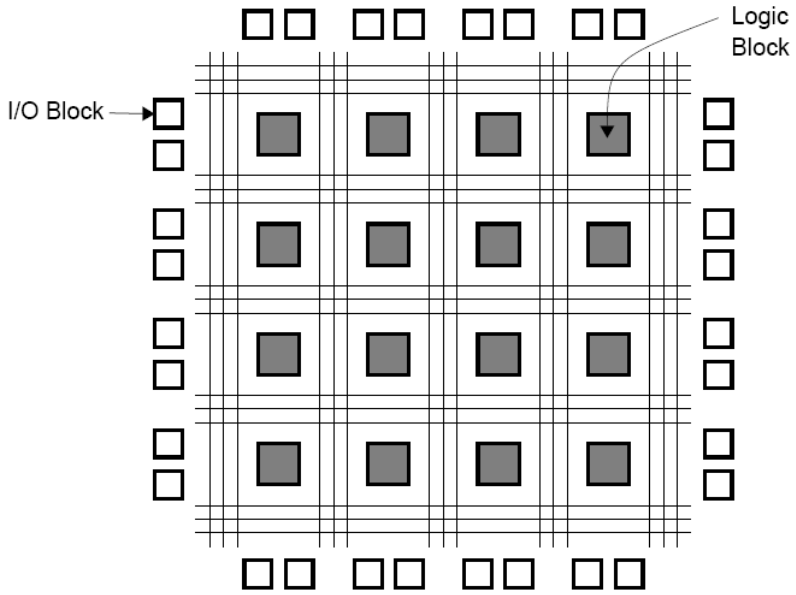
Com o avanço da computação no decorrer dos anos e o constante aumento do tamanho e requisitos das aplicações computacionais, foi natural o aparecimento de problemas que necessitavam de mais poder de processamento do que uma abordagem puramente em software poderia oferecer.

Para esse tipo de problema surgiram os aceleradores em hardware. Essa nova classe de elementos de processamento é desenvolvida para resolver um único problema específico, podendo atingir altos patamares de desempenho.

Dentre esses processadores de uso específico se destacaram os *Digital Signal Processors*(DSPs) para processamento de sinais e mais a frente os *Application Specific Integrated Circuits* (ASICs) que resolvem problemas mais genéricos. Os dispositivos reconfiguráveis apareceram quando o desenvolvimento dos ASICs se tornou muito caro para pequenos problemas.

Os hardwares reconfiguráveis do tipo *Field Programable Gate Arrays* (FPGAs) são dispositivos capazes de serem configurados como um ASIC atingindo custos bem menores, porém mantendo enquanto tentam manter o nível de paralelismo e desempenho.

Figura 2.1 - Um tipo de arquitetura de FPGA.



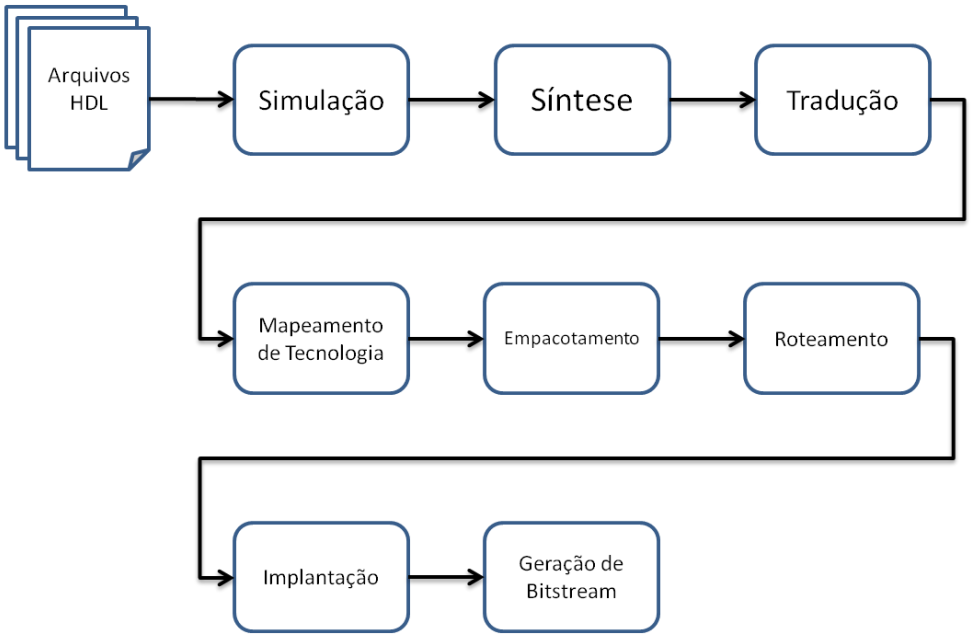
Fonte: BROWN e ROSE, 1996

A Figura 2.1 mostra um tipo de comumente usada nas FPGAs (BROWN e ROSE, 1996), as unidades centrais chamadas Logic Block variam de fabricante para fabricante, mas basicamente são formadas por Lookup Tables (LUTs), unidades de memória e alguns outros elementos lógicos, a interconexão entre os elementos é feita através das linhas intermediárias entre os Logic Blocks. Os I/O Blocks são responsáveis pela comunicação externa do dispositivo. A configuração dos componentes na FPGA é feita através de elementos de memória SRAM que são configurados através de um comando serial chamado bitstream.

2.1.1 Fluxo de Desenvolvimento de hardware para FPGAs

As etapas de desenvolvimento de arquiteturas de hardware para FPGAs é bem semelhante independentemente do fabricante do dispositivo, embora alguns passos possam ser modificados ou customizados ficando de acordo com as ferramentas de desenvolvimento fornecidas.

Figura 2.2 - Etapas de desenvolvimento de hardware para FPGA.



Fonte: Elaborada pelo Autor.

Como mostrado na Figura 2.2, a primeira etapa é a concepção da descrição dos componentes de hardware. Nessa etapa normalmente são usadas linguagens de descrição de hardware (HDL) como Verilog ou VHDL, para descrever o comportamento e os componentes da arquitetura de hardware desejada.

A segunda etapa consiste no Teste e Simulação da arquitetura desenvolvida, essa etapa é opcional, mas normalmente é uma etapa importante por ser capaz de identificar problemas lógicos antes da execução das etapas mais demoradas do processo.

A Síntese, logo após a simulação da arquitetura, é a terceira etapa. Nesta fase os arquivos de descrição de hardware são transformados em uma descrição de mais baixo nível em termos de *LUTs*, registradores e elementos básicos de lógica. Essa representação é transformada para uma representação, normalmente dependente do fabricante, denominada *netlist*.

Após a etapa de Síntese, é feita a Tradução onde todos os arquivos individuais de descrição são unidos em uma única *netlist*, porém em um formato dependente do fabricante. Em seguida, a etapa de Mapeamento de Tecnologia recebe a *netlist* gerada e a mapeia em outra, que utiliza componentes de hardware

auxiliares, dependentes da plataforma de hardware onde a arquitetura será configurada.

Na etapa de Empacotamento a *netlist* é processada para que seus componentes básicos sejam agrupados em *Logic Blocks*. As etapas de Roteamento e Implantação definem a localização física dos recursos utilizados bem como quais interconexões serão feitas entre eles.

Por fim todos os arquivos gerados são utilizados para gerar um *bitstream* que é uma codificação serial dos resultados obtidos nas etapas de empacotamento, roteamento e implantação. Nesse ponto a arquitetura está pronta para ser implantada no dispositivo.

2.1.2 Reconfiguração em Tempo de Execução

Com a evolução dos sistemas reconfiguráveis surgiram os componentes com a capacidade de reconfiguração em tempo de execução. Pelo ponto de vista dessa característica, os dispositivos de hardware reconfigurável podem ser divididos em função da sua capacidade de reconfiguração: estática ou dinâmica.

Dispositivos de reconfiguração estáticos são aqueles configurados antes da operação começar. Tais dispositivos precisam ser parados e reiniciados caso uma nova configuração seja necessária. Já nos dispositivos de reconfiguração dinâmica as configurações são feitas antes e durante a operação.

A reconfiguração dinâmica pode ser dividia em mais dois grupos: reconfiguração orientada aos dados e reconfiguração orientada ao *host*. A reconfiguração orientada a dados é observada em um dispositivo de hardware que se reconfigura dependendo diretamente de dados processados. Por exemplo, um processador de vídeo mudar o algoritmo de decodificação caso encontre um cabeçalho informando um tipo de mídia diferente do que está configurado para processar.

Já no tipo de reconfiguração orientado ao *host*, a mesma é controlada por um processador. Esse processador tem acesso à memória de configuração do dispositivo e pode mudar a configuração do mesmo a qualquer momento através de software.

2.1.3 Algumas técnicas de reconfiguração dinâmica

Dado que a reconfiguração parcial é possível, muitas técnicas foram desenvolvidas visando esse tipo de design (KAO, 2005). As técnicas de reconfiguração pressupõem que o hardware já foi desenvolvido objetivando uma plataforma de reconfiguração dinâmica.

2.1.3.1 Reconfiguração Parcial Baseada em Módulos

Dentre as técnicas de reconfiguração, a baseada em módulos (SEDCOLE, BLODGET, *et al.*, 2006) (MERMOUD, 2004) consiste em uma técnica simples porém que restringe a etapa de Projeto em muitos aspectos. A técnica consiste em desenvolver vários projetos separados para cada arquitetura de hardware que se deseja configurar na plataforma. Cada módulo deve ser desenvolvido de forma separada, a comunicação entre os módulos deve ser feita através de componentes estáticos que podem ser colocados na plataforma no início da operação.

No caso em que mais de um módulo estará configurado na plataforma de hardware ao mesmo tempo, o desenvolvedor precisa ter certeza de que os módulos não se sobrepõem na etapa de Implantação, e que o roteamento dos componentes de comunicação da arquitetura não sobreponha as áreas destinadas aos componentes de hardware que possam ser configurados na plataforma.

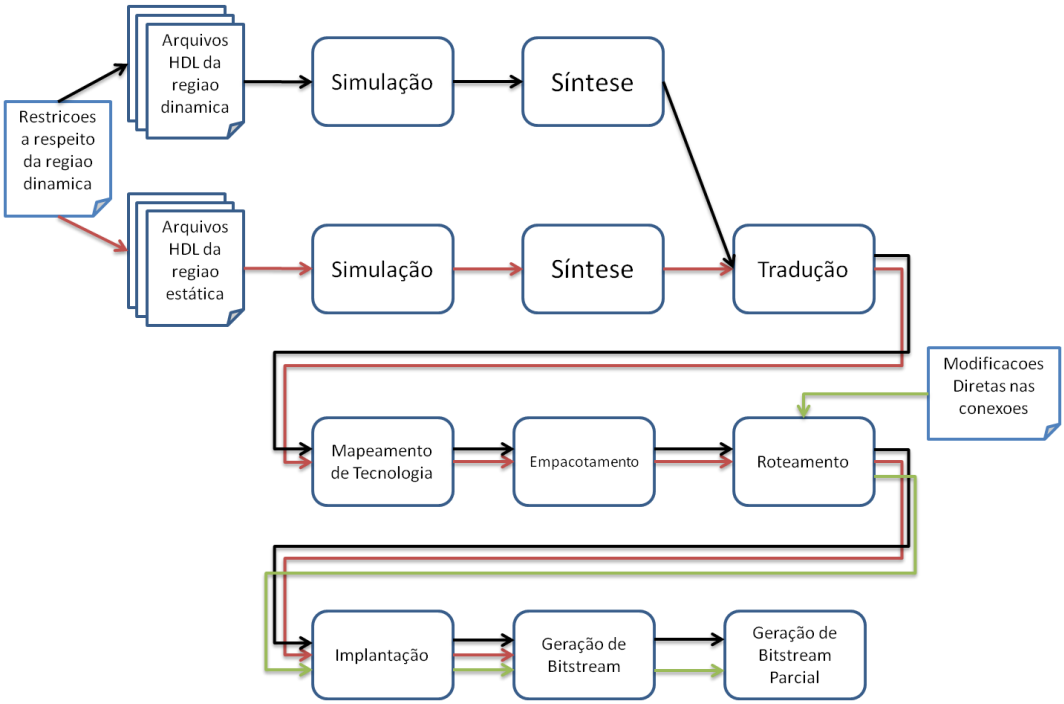
Uma vez que a porção estática tenha sido configurada, os outros módulos podem ser configurados à medida que forem necessários. Respeitando as restrições anteriormente citadas.

A vantagem dessa solução está na falta de necessidade de manipulação do *bitstream,* pois cada arquitetura de hardware é desenvolvida separadamente respeitando restrições espaciais. Porém, esta abordagem possui maior tempo de reconfiguração em relação a outras soluções e não possui uma interface sólida de testes que permita a localização de problemas na execução antes de ser implantada no dispositivo.

2.1.3.2 Reconfiguração Parcial Baseada em *Bitstream* Parcial

Esse método consiste na geração de *bitstreams* que contém apenas a diferença entre o *bitstream* presente na plataforma e o *bitstream* que representa a nova arquitetura a ser configurada (ETO, 2007).

Figura 2.3 - Projetos de hardware reconfigurável baseado em *bitstream* parcial. Linhas pretas apresentam o projeto baseado em Regiões Reconfiguráveis; Linhas Verdes representam o projeto baseado em pequenas modificações na arquitetura; Linhas vermelhas representam o fluxo de projeto convencional.



Fonte: Elaborada pelo Autor.

Como mostrado na Figura 2.3, as ferramentas de desenvolvimento suportam, no geral, dois tipos de projetos que geram *bitstream* parciais. No primeiro, são delimitadas regiões na FPGA denominadas Regiões Reconfiguráveis, essas regiões delimitam os algoritmos de roteamento para geração de no mínimo dois *bitstreams*: Um estático representando toda a região restante fora da área reconfigurável; outro dinâmico representando a porção de FPGA dentro da região reconfigurável. O *bitstream* estático é configurado apenas uma vez na FPGA, enquanto os *bitstreams* das regiões reconfiguráveis podem ser configurados quantas vezes forem necessárias.

A segunda forma de geração de *bitstream* parcial consiste em modificações no resultado da arquitetura roteada, uma vez que o roteamento é feito, podemos

modificar quais elementos se conectam de forma a mudar o funcionamento da arquitetura de hardware, sem mudar o posicionamento dos componentes. Esse método é mais rápido, em termos de geração do *bitstream* final, porém difícil de aplicar, uma vez que as modificações no roteamento precisam ser feitas sem auxílio das ferramentas de roteamento.

Independente de qual método seja usado, uma das principais vantagens dessa metodologia é o suporte direto provido pelas ferramentas de desenvolvimento. Durante a especificação da arquitetura de hardware, o desenvolvedor irá dividir o ciclo de execução em modos de operação. Cada modo será composto de alguma descrição de hardware específica, qualquer reconfiguração consistirá da mudança de um modo para outro.

A partir da lista de todas as transições de modo de operação possíveis, uma ferramenta gera os *bitstreams* parciais que levam o hardware de um modo de operação para outro fazendo entre um estado de configuração e o outro.

Além disso, a utilização dos *bitstream* parciais reduz consideravelmente a quantidade de informação necessária na reconfiguração do dispositivo. Ao invés de guardar descrições inteiras de hardware a aplicação, esta guarda apenas a diferença entre as configurações. Em adição não existe a preocupação com a distribuição espacial dos componentes de hardware uma vez que o *bitstream* gerado é sempre checado e para garantir que não haverá sobreposição de componentes durante a reconfiguração.

2.1.4 Reconfiguração Baseada em plataformas Multi Contexto

Uma das maneiras mais efetivas de modificar o hardware em execução em um dispositivo reconfigurável é a metodologia Multi-Contexto (LING e AMANO, 1993) (TRIMBERGER, CARBERRY, *et al.*, 1997). A palavra contexto, nesse caso, diz respeito ao hardware atualmente operando no dispositivo. Dessa forma a expressão Hardware Multi-Contexto exprime a presença de várias arquiteturas de hardware armazenadas no dispositivo, porém apenas uma permanece em execução.

Embora possa parecer semelhante com o conceito básico de reconfiguração dinâmica, a diferença principal, entre os hardwares multi-contexto e as metodologias anteriormente apresentadas, é que múltiplas arquiteturas de hardware são

armazenadas na inicialização da plataforma de hardware. Essas arquiteturas serão chaveadas de acordo com um conjunto de regras. Essas regras podem ser implementadas em um processador host, através de eventos observados no processamento dos dados ou através de um controle temporal implementado na própria plataforma de hardware.

Em uma arquitetura multi-contexto a mudança entre o hardware em execução e um hardware que esteja preparado em um contexto inativo é muito eficiente. Dessa forma os tempos de reconfiguração são muito pequenos.

A principal vantagem dessa metodologia sobre as outras é o tempo de configuração, porém a necessidade de fazer a o carregamento das arquiteturas de hardware *offline* do sistema restringe bastante as arquiteturas reconfiguráveis que visam a flexibilidade do hardware em execução nas plataformas.

2.2 SISTEMAS DE SOFTWARE ORIENTADOS A COMPONENTES

O termo Componente de Software é bastante ambíguo em seu significado, sendo tratado de diferentes formas dependendo do autor e do trabalho (KLABUNDE, 1999). Segundo o trabalho mostrado em (SZYPERSKI, 2002) componentes de software tem as seguintes características:

- São entidades de desenvolvimento independente: Implica que podem ser desenvolvidos separadamente do seu ambiente de aplicação ou de outros componentes.
- São entidades combináveis: Um componente pode ser desenvolvido por terceiros e ser integrado em um ambiente de execução qualquer com o auxílio de sua documentação e suas interfaces.
- Não possui estado observável externamente: Não existem diferenças, em termos de interfaces externas, entre diversas instâncias do mesmo componente.

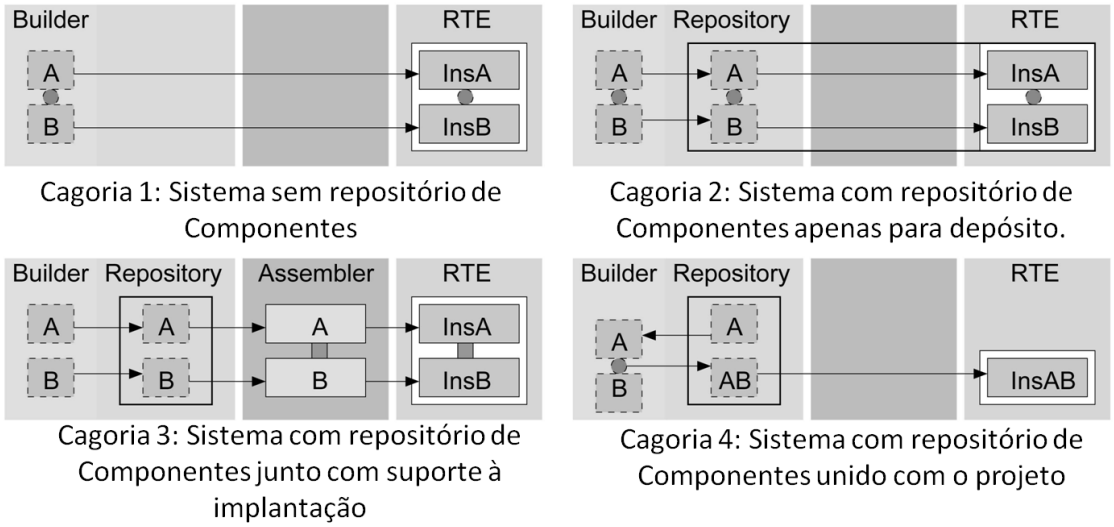
Uma das principais vantagens do uso de componentes de software é a reutilização de código confiável. Dessa forma é possível construir componentes maiores utilizando o agrupamento de vários menores e agregar também a confiança desses outros de forma a todo o sistema possuir um determinado crédito em relação

a um sistema desenvolvido a partir de metodologias que não usufruem da reutilização.

2.2.1 Modelos de Software Baseado em Componentes

Os modelos de software baseado em componentes (LAU e WANG, 2007) agrupam arquiteturas que propõem soluções para o tratamento dos componentes de software, tais como: quais interfaces básicas devem prover; o ciclo de vida dos componentes; e elementos conectores comuns aos componentes.

Figura 2.4 - Tipos de modelos de software classificados quanto ao tipo de repositório.



Fonte: (LAU e WANG, 2007)

Esses modelos apresentam uma entidade em comum que representa uma forma de localizar e retornar componentes de software presentes em uma base de dados. Esse elemento, quando não vem na forma de um repositório, é representado através de descritores de componentes que devem ser fornecidos para permitir a procura. A Figura 2.4 mostra uma classificação possível desses modelos de componentes quanto ao tipo de repositório de componentes, seja qual for o modelo o objetivo é o mesmo: encontrar um determinado componente em meio a um conjunto, objetivando compor um sistema maior.

2.3 REPOSITÓRIOS DE COMPONENTES DE SOFTWARE

Um repositório de componentes é um mecanismo que possibilita o armazenamento, localização e recuperação de componentes de software de forma a promover o reuso dos mesmos (LUCRÉCIO, PRADO e SANTANA, 2004). Abaixo, encontram-se listadas algumas das principais vantagens relacionadas à utilização de repositórios de componentes (PEREIRA, 2000):

- Menor custo de busca - permitir que a busca de componentes possa ser mais eficiente, diminuindo no sistema usuário a carga envolvida na recuperação.
- Suporte ao desenvolvimento em larga escala - em função de um menor custo na recuperação, o acesso aos componentes desenvolvidos é facilitado, com isso, mais componentes são acessados;
- Padronização de armazenamento e recuperação - Utilizando-se padrões para o acesso e recuperação dos componentes permite-se que o repositório possa ser acessado por diferentes usuários.

2.3.1 Repositórios Ativos de Componentes de Software

Repositórios ativos de componentes são repositórios capazes de fornecer informações sobre componentes de software para os desenvolvedores de sistemas sem que eles necessitem formular requisições (YE, 2001).

Tais repositórios podem ser integrados aos ambientes de desenvolvimento de sistemas orientados a componentes e, com base nas tarefas realizadas pelos desenvolvedores, fornecer informações sobre componentes que podem ser utilizados para compor o sistema em desenvolvimento. Essa abordagem permite que os desenvolvedores façam uso de componentes por eles desconhecidos e que o custo do reuso seja diminuído.

Deve ser destacado que o conceito de repositório ativo, proposto inicialmente por Ye (YE, 2001), é utilizado apenas durante a fase de desenvolvimento de sistemas orientados a componentes. Uma vez em execução, os sistemas não mais fazem uso do mecanismo de entrega ativa de informação do repositório.

CAPÍTULO 3
CENÁRIOS DE USO

Sistemas de hardware são desenvolvidos de forma menos flexível que sistemas de software. Ao desenvolver um hardware é muito comum especificar parâmetros que devem ser estritamente seguidos durante a operação. Tais restrições de operação podem ser vistas como atributos do contexto onde o hardware está inserido, leituras desses atributos podem ser feitas e serem usadas, por exemplo, a fim de certificar a corretude do hardware em questão. Essas leituras são usadas apenas para monitoramento em sistemas estáticos, mas podem ser usadas como informação útil para sistemas dinâmicos.

No escopo deste trabalho as variáveis relevantes ao funcionamento do hardware formam um conjunto de tuplas denominado Contexto de Operação da aplicação, em conjunto com a Arquitetura de Gerenciamento proposta, esse conjunto de variáveis e suas condições de funcionamento compõem um sistema orientado ao contexto. Leituras do contexto de operação, cruzadas com as condições de funcionamento especificadas para o sistema, podem ser usadas para disparar modificações na estrutura do hardware visando adaptação a novas faixas de operação de forma dinâmica ou aperfeiçoar sistemas em execução de forma a adaptá-los melhor ao seu contexto atual.

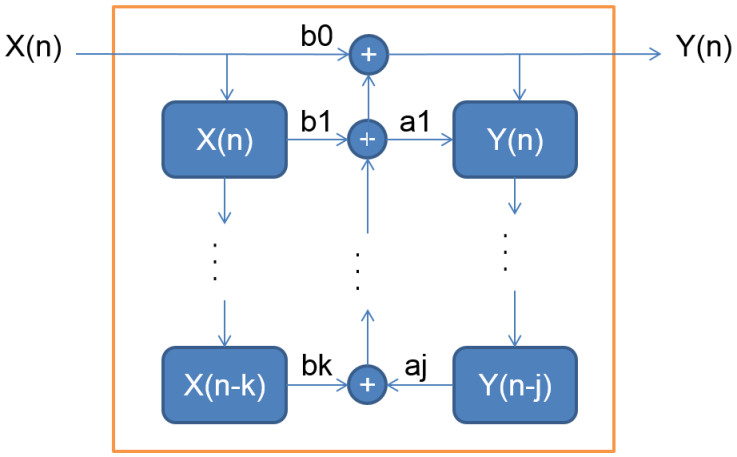
Tendo em vista esses pontos, este capítulo tem por objetivo expor cenários que evidenciam a utilidade de uma arquitetura que monitore e gerencie o hardware executando em Aplicações Embarcadas de forma a adaptá-lo ao Contexto de Operação. Os cenários serão expostos usando uma descrição dos problemas envolvidos, um exemplo de modelagem para solução desses problemas usando uma arquitetura de monitoramento de contexto e os possíveis benefícios que podem ser alcançados usando uma arquitetura como a proposta nesta Tese.

3.1 FILTROS DIGITAIS DE SINAIS

Filtros digitais são aplicações comuns quando se pensa em sistemas em hardware. Um dos principais motivos que levam filtragem a ser feita em hardware é

o requisito de banda muito alto e a facilidade de implementação de filtros devido a sua estrutura básica. Filtros são basicamente unidades que processam sinais discretos.

Figura 3.1 - Estrutura básica de um filtro digital.



Fonte: Elaborada pelo Autor.

A estrutura básica dos filtros digitais, apresentada na Figura 3.1, mostra uma característica importante para os que podem ser gerenciados pela arquitetura apresentada nesta tese: sua arquitetura é composta por vários submódulos, que são independentes e podem ser reestruturados para obtenção de diferentes resultados. Além disso, possui uma quantidade bem definida de entradas e saídas, onde, independentemente da interconexão e tipo de subcomponentes, essa quantidade não varia.

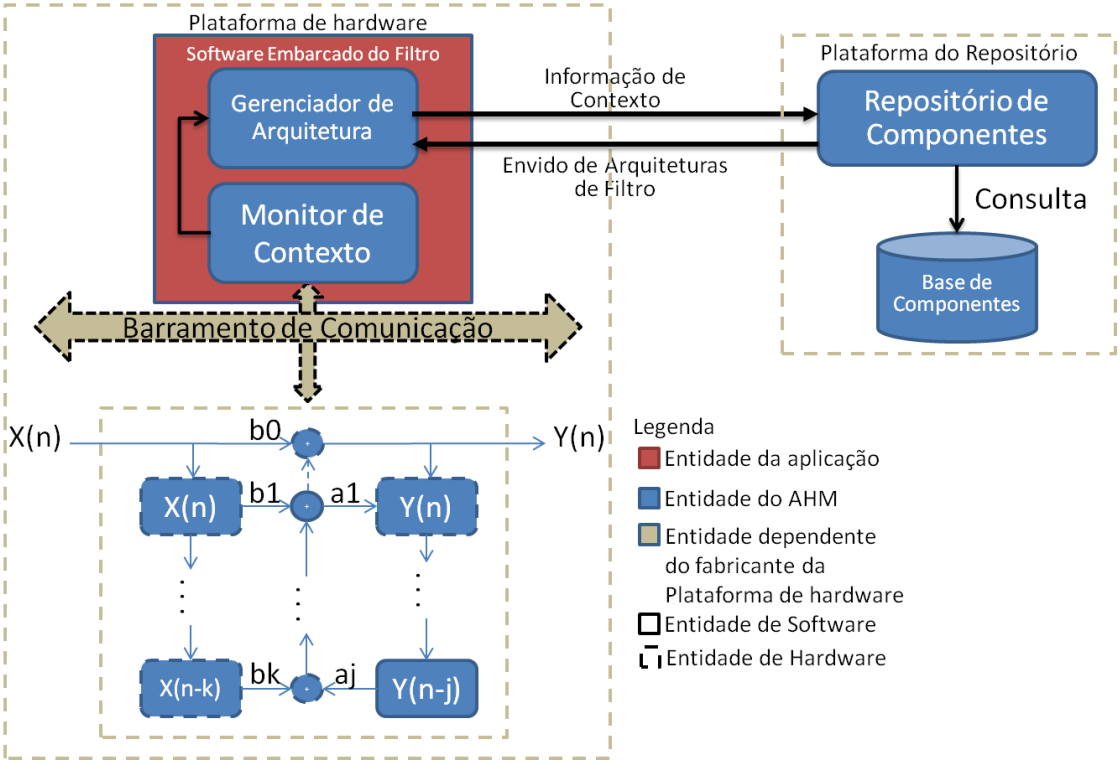
Um problema comum buscado na arquitetura de filtros é sua capacidade de adaptação. Um filtro com arquitetura fixa é capaz de lidar com um sinal restrito a dados tipos de ruído que são estudados e medidos com o intuito de melhorar a qualidade do sinal filtrado. Dessa forma o projeto de filtros envolve passos que devem levar em conta muitos aspectos a respeito do sinal desejado, embora muitos desses aspectos sejam aleatórios e variem com o tempo.

A eficiência de um filtro pode ser prejudicada pela variação acentuada de muitos parâmetros, um parâmetro comum é o valor da relação entre sinal e ruído ou *Signal Noise Ratio* (SNR). Filtros comumente são projetados para um determinado limite de SNR, porém devido a condições diversas a potência do ruído pode ser grande demais, ou o tipo de filtro pode simplesmente não ser adaptado ao ruído

encontrado em uma dada faixa de frequência. Para casos como esses alguns filtros podem usar estruturas dinâmicas como os filtros adaptativos (HAYKIN, 1996).

Um filtro adaptativo pode ser desenvolvido para modificar os parâmetros ou a quantidade de atrasos na arquitetura do filtro. O processo de filtragem dinâmica envolve um motor capaz de modificar os pesos e em alguns casos a estrutura do filtro com o passar do tempo. No entanto essa modificação pode não ser suficientemente rápida ou não resolver o problema no pior caso. Em contrapartida para muitos dos casos em que um filtro não é ideal existe uma outra configuração ou um outro filtro que se adapta melhor ao cenário, dessa forma o caso ideal seria modificar o filtro para se adaptar a cada condição de operação. Nesse aspecto o sistema proposto nesta tese possibilita uma solução mais genérica para o caso apresentado.

Figura 3.2 - Arquitetura de um sistema de filtro digital sensível ao cotexto.



Fonte: Elaborada pelo autor.

A arquitetura proposta na Figura 3.2, mostra quatro componentes básicos, um monitor de contexto, um gerenciador de arquitetura do filtro, um componente repositório além de uma base de dados de componentes. Esses componentes possibilitam a implementação de um sistema de filtragem adaptativo, onde novas

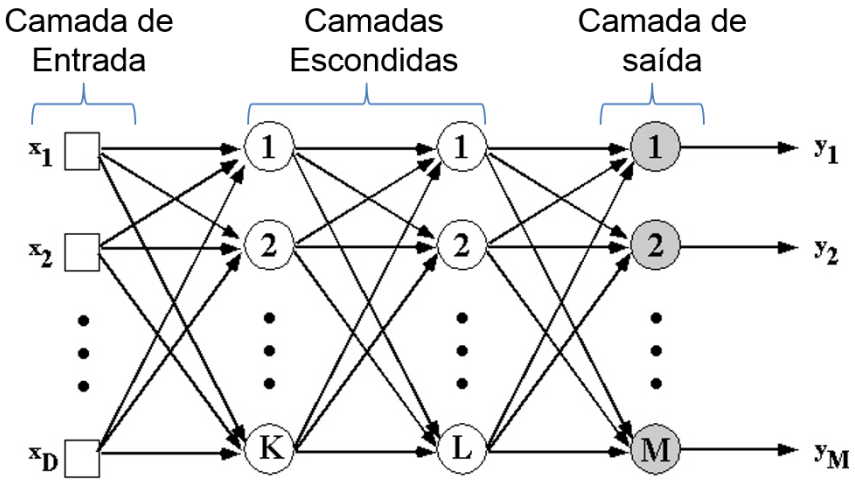
arquiteturas de filtros podem ser geradas usando a comunicação entre o Gerenciador de Arquitetura do filtro e o Repositório de componentes. Novos tipos de filtro podem ser adicionados na base de componentes ou mesmo algum evento importante, como a presença de algum tipo de ruído não esperado, pode acarretar uma mudança na arquitetura do filtro em execução.

Assim o sistema pode ser reconfigurado vide variações no contexto de operação. As variáveis que compõem o contexto de operação podem variar dependendo da aplicação onde o filtro está inserido. Para o caso básico apresentado, apenas os valores de Y(n) e X(n), representados na figura, além de seus valores atrasados no tempo são levados em consideração. Uma versão mais complexa do sistema poderia inserir métricas a respeito de outras grandezas que tenham influencia na qualidade do sinal X(n).

3.2 REDES NEURAIS ARTIFICIAIS

As Redes Neurais Artificiais (RNA) compõem um domínio de aplicações que cada vez mais são implementadas em hardware. RNAs são aplicadas em muitos domínios diferentes tais como implementação de funções (HAYKIN, 1998) matemáticas, classificadores, identificação de plantas de controle, entre outros.

Figura 3.3 - Arquitetura básica do *Multi Layer Percptrom* (MLP).



Fonte : (HAYKIN, 1998).

A Figura 3.3 mostra uma arquitetura do MLP um tipo clássico de RNA. Embora existam muitas variações da estrutura básica, o formato geral é composto sempre de componentes de processamento, os Neurônios, e interconexões entre

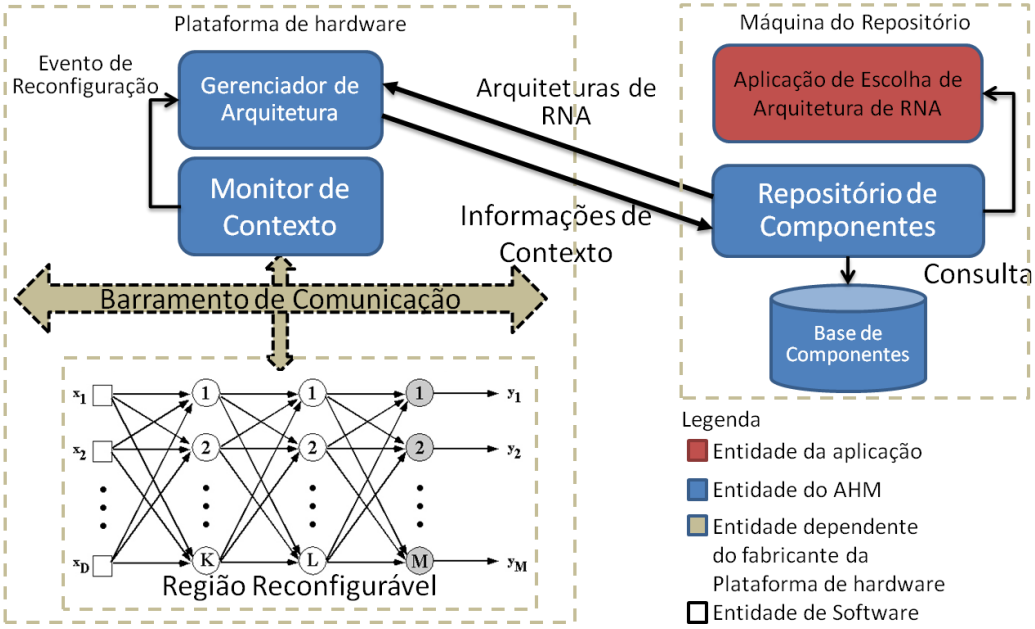
eles denominadas Sinapses. A arquitetura de interconexão, o processamento realizado nos neurônios, a disposição dos neurônios nas camadas e o algoritmo de modificação dos pesos sinápticos caracteriza o tipo de RNA.

A implementação desses sistemas em hardware é normalmente complicada devido ao número de componentes, interconexões internas e a extensa quantidade de operações de ponto flutuante e/ou fixo. Porém a alta capacidade de paralelização das RNAs é uma característica a favor das soluções em hardware.

Uma característica interessante do domínio das RNA é o fato de que existem muitas arquiteturas possíveis para resolver determinado problema. Dessa afirmação surge o campo de aplicações que abrangem otimização de redes neurais objetivando reduzir os custos da rede necessária para resolver dado problema, em termos de componentes e interconexões.

A implementação de redes neurais sensíveis ao contexto é um pouco mais restrita do que a implementação de filtros. Para aplicar uma RNA na solução de um problema primeiro é executado um treinamento *offline* de forma a descobrir quais os pesos e arquitetura que mais se adéquam ao problema a ser resolvido. Uma vez terminado o treinamento a rede pode ser implantada em um componente de hardware e entrar em operação.

Figura 3.4 - Exemplo de arquitetura de Rede Neural sensível ao contexto.



Fonte: Elaborada pelo autor.

Para o caso da aplicação da arquitetura em redes neurais, a arquitetura da Figura 3.4, apresenta quatro componentes básicos: um monitor de contexto; um gerenciador de arquitetura; um repositório de componentes com sua base de dados; por fim, uma aplicação executando próxima ao repositório de componentes que processa e gera novas arquiteturas de rede neural.

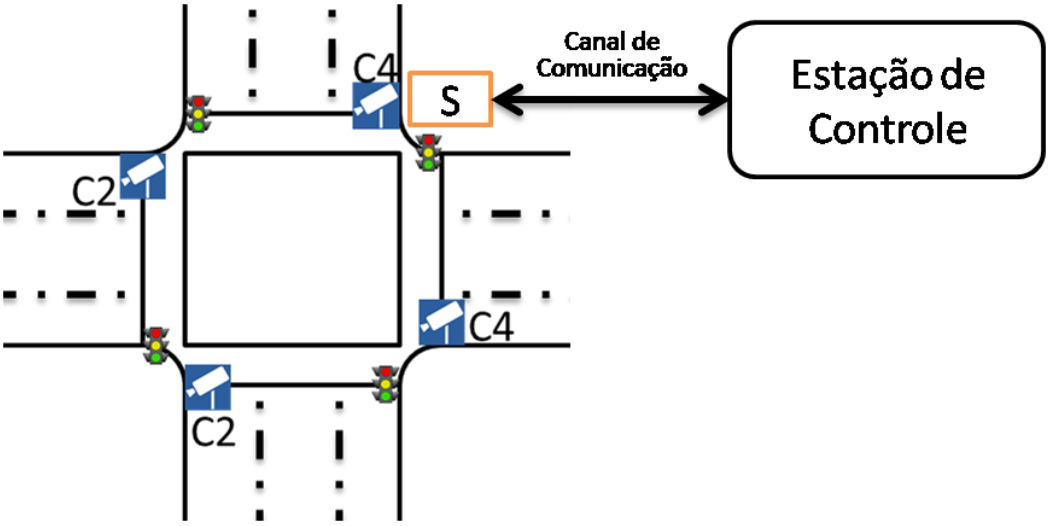
As variáveis do contexto de operação podem ser medidas em termos das entradas e saídas da rede neural comparadas com as dos valores esperados para cada uma, outras variáveis do contexto de operação podem ser dependentes de aplicação como, por exemplo, o valor SNR se a rede estiver sendo para filtragem de sinais.

Nesse caso, existe uma aplicação que executa junto ao repositório de componentes, otimizando a arquitetura que executa no sistema gerenciado. Como mencionado existem muitas arquiteturas de RNA que podem resolver dado problema, nesse caso os eventos de modificação de arquitetura podem não ser suficientes para definir qual rede deve ser usada, portanto foi adicionada uma aplicação que usa as APIs disponibilizadas pelo Repositório para realizar a reconfiguração do sistema de hardware uma vez que novas arquiteturas de rede, que melhor se adaptem às condições do sistema, sejam encontradas.

3.3 SISTEMAS DE PROCESSAMENTO DE IMAGENS

Algumas aplicações não possuem uma dinamicidade tão grande quanto os Filtros Digitais ou Redes Neurais, um exemplo são aplicações que utilizam processamento digital de imagens. De acordo com a literatura relacionada (HUIYUAN, JUN e ZHE, 2007) (YAN, ZHANG e LALA, 2009), um sistema de câmeras de trafego de veículos como o exemplificado na Figura 3.5 combina um conjunto de sistemas que variam desde algoritmos de processamento de imagens até sistemas de segurança complexos.

Figura 3.5 - Exemplo de cruzamento monitorado por câmeras de transito.

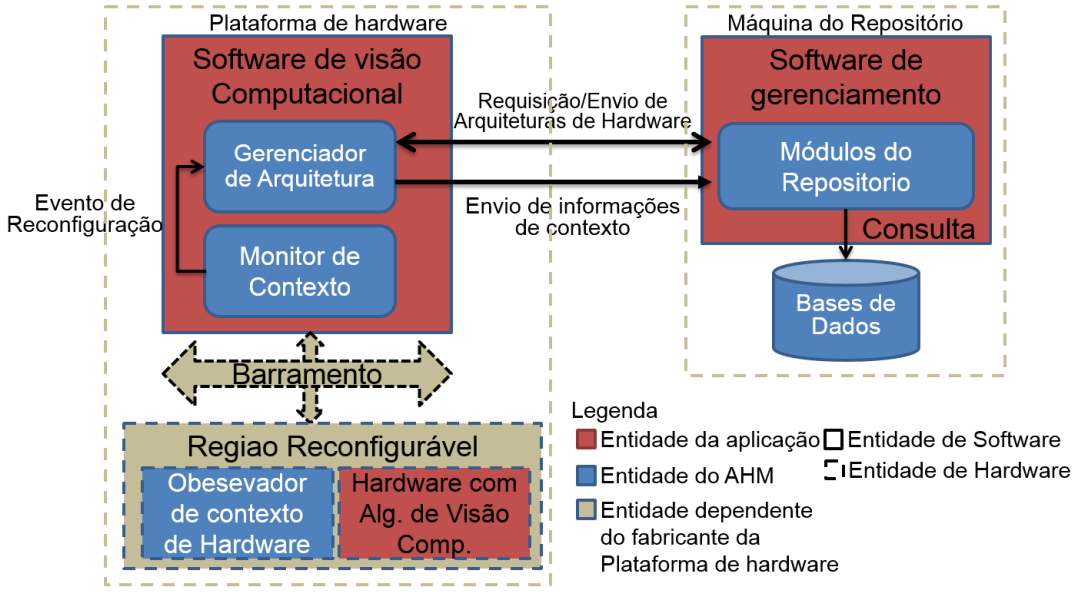


Fonte: Elaborada pelo autor.

Para simplificar o exemplo, assumiremos que o sistema de monitoramento precisa apenas contar quantos carros param em cada um dos semáforos uma vez que ele está fechado. Para esta aplicação, pode-se considerar que alguns algoritmos de processamento de imagens são necessários. Podemos verificar ainda que possam existir, no mínimo, diferentes condições de iluminação para cada uma das câmeras, devido às diferentes posições de cada uma delas.

Dito isto, podemos inferir que cada uma das câmeras precisará de um algoritmo especialmente calibrado, para contornar as suas condições específicas. Uma possível solução seria instalar as câmeras e calibrar os algoritmos manualmente para cada uma delas. Porém uma solução de adaptação automática seria mais útil, pois aceleraria o tempo de instalação e funcionamento do sistema.

Figura 3.6 - Sistema autonômico de processamento de imagens.



Fonte: Elaborada pelo autor.

A Figura 3.6 apresenta um exemplo de como um sistema autonômico de gerenciamento poderia ser construído para adaptar os algoritmos de processamento de imagens usados nesse cenário. O cenário dessa vez prevê que uma região de hardware reconfigurável, onde alguns algoritmos relativos à aplicação serão implementados, esteja presente.

Mais uma vez o monitor de contexto e o gerenciador de arquitetura são usados, do lado do sistema gerenciado, porém um novo componente, visando monitorar variáveis de contexto diretamente relacionadas com componentes de hardware foi adicionado, chamado de Observador de Contexto de Hardware. No lado do repositório, uma aplicação de gerenciamento executa junto aos módulos do mesmo.

Nesse caso duas aplicações foram desenvolvidas, uma usando as APIs do Repositório e outra usando as APIs do cliente para a reconfiguração do hardware. Nesse sistema as APIs no cliente são usadas para reconfigurar o sistema, adaptando-o às condições de iluminação, enquanto as APIs no Repositório podem ser usadas para realizar otimização e atualização do sistema quando necessário.

Variáveis de contexto como as condições de iluminação em cada uma das câmeras processadas podem ser usadas para aperfeiçoar o sistema de acordo com a localização de cada uma delas. Outra variável poderia ser uma medida de

desempenho usando uma imagem de calibração, que poderia ser usada para adaptar o sistema a um ponto em que funcione de forma mais aceitável.

3.4 VEÍCULOS AÉREOS NÃO TRIPULADOS (VANTS)

VANTs estão dentre as áreas mais pesquisadas na área de Robótica (IEEE, 2012). Tanto autônomos quanto controlados por rádio, os VANTs são projetos de engenharia bastante complexos. No projeto de VANTs, parâmetros como área da aeronave, carga útil e missão a ser realizada precisam ser levadas em consideração. Tais parâmetros podem decidir, no pior caso, até sobre a viabilidade do projeto.

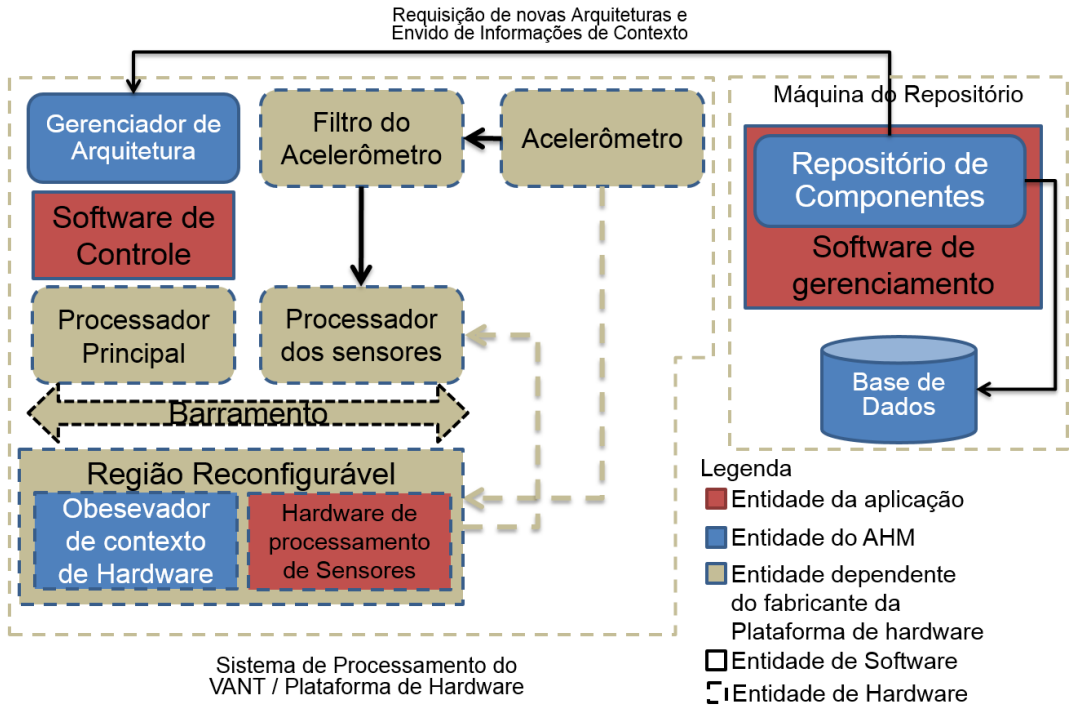
Nos VANTs são comuns condições de operação mais extremas, de forma que o hardware usado nesses veículos precisa ser estudado de forma mais cuidadosa. Problemas como ruído advindo dos motores e a distância da unidade de controle manual podem acarretar falhas significativas na missão do veículo.

A utilização de um sistema orientado à contexto em VANTs é vantajosa em vários aspectos. Um aspecto bem visível relaciona-se com a missão a ser realizada pelo veículo. Dispositivos de hardware ocupam espaço físico e demandam um determinado local de acomodação que deve ser testado para evitar falhas. O espaço físico ocupado por um hardware pode ser decisivo no dimensionamento da aeronave. Nesse caso um VANT dimensionado para determinada missão que use alguns processadores em hardware específicos precisaria ser reprojetado se fosse feito para outra missão que usasse outros tipos de algoritmos em hardware.

Nesse caso a substituição desses processadores por processadores reconfiguráveis que possam ser modificados de acordo com a missão da aeronave é uma solução interessante. A arquitetura apresentada nessa tese se adéqua bem a esse tipo de aplicação, embora nenhum elemento de contexto tenha sido mencionado, a arquitetura proposta é capaz de gerenciar o hardware presente na aeronave e modificá-lo para se adaptar à nova missão.

Uma situação que é dependente do contexto de operação pode ser observada no caso em que ocorra uma falha de funcionamento de algum dos módulos de processamento da aeronave. Exemplos de módulos que podem falhar são elementos de controle dos motores, filtros de sensores, coprocessadores, dentre outros.

Figura 3.7 - Exemplo de cenário onde uma arquitetura sensível ao contexto é implantada em um VANT.



Fonte: Elaborada pelo autor.

A Figura 3.7, mostra como um sistema de filtragem do sinal advindo do acelerômetro pode ser substituído no caso de falha desde que todas as entradas estejam conectadas ao dispositivo reconfigurável. Os componentes nesse caso são representados de forma mais específica em uma plataforma de VANT com suporte a hardware reconfigurável. O gerenciador de arquitetura, executando no processador configura um componente de hardware junto com o Observador de Contexto de Hardware na região reconfigurável no caso em que alguma falha no processador dos sensores ocorra. O restante dos componentes, o repositório de Componentes e sua base de dados, executam fora da plataforma, gerando otimizações, atualizações e monitoramentos.

Um ponto importante a salientar é de que para que a arquitetura funcione com muitas partes do avião, o dispositivo reconfigurável deve estar ligado a muitas entradas físicas da aeronave, de forma que todas as entradas e saídas que vão e saem de um dispositivo substituível devem também ser ligadas ao dispositivo reconfigurável.

As variáveis de contexto do sistema do VANT podem variar dependendo do objetivo do sistema reconfigurável. Por exemplo, para um sistema de reposição de componentes falhos as variáveis de contexto podem ser um conjunto de valores que representam a presença ou ausência dos módulos de hardware a serem substituídos. Os eventos que dispararíam uma reconfiguração, nesse caso, seriam a detecção da falha de algum dos componentes substituíveis.

CAPÍTULO 4
TRABALHOS RELACIONADOS

Embora esta tese seja focada na área de Sistemas Autonômicos, existem alguns trabalhos que não mencionam essa grande área, mas mesmo assim possuem uma fraca ou forte relação com a arquitetura proposta nesta tese. Portanto, para melhor entendimento das relações entre os trabalhos apresentados e a tese proposta, iremos dividir os trabalhos encontrados na literatura em grupos, apresentando mais detalhadamente trabalhos que possuem maior semelhança.

Para melhor entendimento os trabalhos serão agrupados em dois grupos, o primeiro grupo abordará ferramentas usadas no auxílio ao fluxo de desenvolvimento de hardware reconfigurável, já o segundo grupo abordará APIs de manipulação de hardware reconfigurável.

4.1 FERRAMENTAS DE AUXÍLIO AO FLUXO DE DESENVOLVIMENTO

Na área de desenvolvimento de hardware reconfigurável o fluxo de desenvolvimento de hardware é a maneira mais clássica de resolver os problemas de alocação e modificação do hardware.

Os fabricantes de dispositivo reconfiguráveis fornecem suas próprias ferramentas de reconfiguração e planejamento da operação do hardware. A xilinx propõe a utilização de sua ferramenta PlanAhead (XILINX CORPORATION) enquanto a Altera utiliza o Quartus (ALTERA CORPORTAION). No entanto ambas as aplicações e fluxos de desenvolvimento requerem muita interferência humana.

No geral as ferramentas que permitem reconfiguração dinâmica trabalham com manipulação e geração de *bitstreams*. Para que uma determinada porção do dispositivo reconfigurável seja gravada é necessário um *bitstream* parcial contendo apenas a diferença entre o *bitstream* atual do dispositivo e o hardware que precisa ser configurado. Para tanto todos os fluxos de desenvolvimento consistem na geração de um *bitstream* principal e vários *bitstream* secundários que serão enviados para os dispositivos quando for necessário.

Uma ferramenta de linha de comando muito referenciada de manipulação de *bitstream* é o PARBIT (HORTA e LOCKWOOD, 2001). Vislumbrando a possibilidade de configurar FPGAs parcialmente surgiu a necessidade de gerar *bitstreams* parciais, ou seja, que não continham toda a informação do hardware, apenas uma fatia da informação que seria gravada no dispositivo de forma que não sobrescrevesse a informação já contida nele.

Para possibilitar a relocação o PARBIT recebe como entrada o *bitstream* desejado e modifica certa parte dele para ocupar apenas certa porção do dispositivo alvo. A configuração pode ser feita em dois modos, no modo "*slice*" e no modo "*block*", caracterizando se o hardware irá ocupar uma fatia ou um bloco da área reconfigurável. Embora a ferramenta possua um objetivo simples, um conhecimento extenso da arquitetura do *bitstream* e da plataforma de destino é necessário para implementação da solução. Uma característica interessante é que toda a informação que a ferramenta precisa é dada via linha de comando, sem necessidade de interface gráfica.

Nessa área existem trabalhos desenvolvidos no sentido de melhorar a geração dos *bitstream* parciais ou mesmo do *bitstream* final da plataforma. Uma arquitetura mais recente proposta para melhorar esse fluxo é o *Tools for Open Reconfigurable Computing-TORC* (COUCH, 2011).

O projeto TORC foi arquitetado como um projeto que precisava modificar designs da Xilinx em todos os níveis possíveis do fluxo de desenvolvimento. Esse projeto envolveu o desenvolvimento de muitas ferramentas para permitir que um sistema embarcado fosse capaz de realizar o trabalho do *toolkit* proprietário da Xilinx. Podendo construir, aperfeiçoar e analisar arquiteturas de hardware.

Sobre a arquitetura provida pelo TORC o OpenPR (SOHANGHPURWALA, 2010) é uma iniciativa de terceiros visando desenvolver um toolkit de ferramentas que implemente todo o fluxo de desenvolvimento de hardware dinamicamente reconfigurável. Possibilitando a melhoria dos pontos onde normalmente as soluções clássicas providas pelos fabricantes possuem complicações.

Uma das principais características focadas no projeto diz respeito à automação do processo de geração de hardware. A automação foi possível através de ferramentas como o GNU Make (GNU), que possibilita o desenvolvimento de scripts externos que controlam o fluxo de geração do hardware e dos *bitstream*.

O nível de automação proposto na ideia geral dessas ferramentas representa um grande esforço em direção da geração automática de hardware. Em nosso trabalho não foram utilizadas as ferramentas propostas no OpenPR ou pelo TORC, pois já havia uma familiarização com as ferramentas da Altera e da Xilinx. A geração de hardware automaticamente através de uma API de software sem intervenção da interface proposta pelos fabricantes é o que diferencia nossa arquitetura das soluções propostas por esses trabalhos.

Na nossa proposta a intervenção humana é necessária para o desenvolvimento da aplicação que irá executar no sistema de controle do hardware, porém após isso ela é desnecessária. De fato tanto as ferramentas propostas pelo OpenPR, TORC ou PARBIT podem ser utilizadas em paralelo ao nosso sistema visando otimizar os designs gerados e proporcionar melhor flexibilidade ao processo de compilação e geração dos *bitstreams*.

No entanto a escolha da utilização das ferramentas dos fabricantes está relacionada com a capacidade da nossa arquitetura acompanhar os novos dispositivos do mercado sem a necessidade de portar muitas ferramentas de geração de hardware.

4.2 APIS DE MANIPULAÇÃO DE HARDWARE RECONFIGURÁVEL

Uma das principais características de nossa solução é a presença de uma API que torna a manipulação do hardware, atualmente configurado no dispositivo, uma atividade automática possibilitando a adaptação do hardware. As soluções anteriormente apresentadas não possuem essas características se limitando à etapa de projeto do hardware.

No quesito API que possibilite modificação do hardware em tempo de execução existe uma solução bastante referenciada a API JBits (GUCCIONE e LEVI, 1998a) (GUCCIONE e LEVI, 1998b). Trata-se de uma API Java proposta pela Xinlix que é usada para manipular o *bitstream* utilizado na configuração das plataformas.

O principal objetivo da JBits é permitir a manipulação do *bitstream* gerado de modo que seja possível alterar pontos desejados e regrava-lo na plataforma. API fornece, para tanto, formas para ler e gravar o *bitstream*, bem como interfaces que permitiam alteração de todos os seus campos. Um aspecto negativo na JBits está

em sua complexidade e dependência da arquitetura alvo para qual o *bitstream* manipulado deve ser gerado.

Embora dependesse fortemente de conhecimento do hardware alvo, outras APIs foram desenvolvidas usando como base a JBits usufruindo da sua habilidade de manipulação do *bistream*. *Java Runtime Rencoifiguration*(JRTR) (MCMILLAN e GUCCIONE, 2000) é uma API que adiciona à JBits o suporte a reconfiguração parcial, removendo uma restrição que obrigava os usuários da JBits a gerar *bitstreams* inteiros toda vez que fosse necessário regravar no dispositivo de hardware.

Ainda usando o suporte da JBits foi desenvolvida a JRoute (KELLER, 2000), projetada para resolver o problema relativo à roteamento que as aplicações que usavam JBits precisavam resolver na criação de novos designs usando a API. A JRoute possibilita várias formas de flexibilização do sistema de roteamento. Através do uso de representação de alto nível, os desenvolvedores não precisam focar em quais bits precisam ser habilitados para que dada configuração gerada.

Embora as ferramentas apresentadas apresentem evoluções, no que se diz respeito à manipulação do hardware através de APIs, a necessidade de conhecer a estrutura binária do *bitstream* e da arquitetura da plataforma de hardware restringe muito as soluções.

Uma solução de mais alto nível, ainda dentro do conjunto das soluções com APIs, é a JHDLBits (POETTER, HUNTER, *et al.*, 2004). Essa solução integrava o mundo das linguagens de descrição de hardware com a geração de *bitstream*. Os componentes de hardware são descritos usando JHDL, (BELLOWS e HUTCHINGS, 1998) uma linguagem de descrição de hardware em Java. Após isso, as descrições eram transformadas em *bitstreams* usando a APIs JBitis. Dessa forma o desenvolvedor de hardware pode focar na arquitetura de hardware, manipulando componentes de hardware. A geração de bitstream e manipulação de bits específicos da plataforma de hardware ficam a cargo da JHDL.

A manipulação no nível de componentes de hardware torna essa última solução próxima do componente de geração de hardware proposto no AHM Repository. Porém a solução proposta pelo AHM é mais flexível por usar as ferramentas do fabricante diretamente, evitando assim o problema de dependência da arquitetura do dispositivo de destino do hardware.

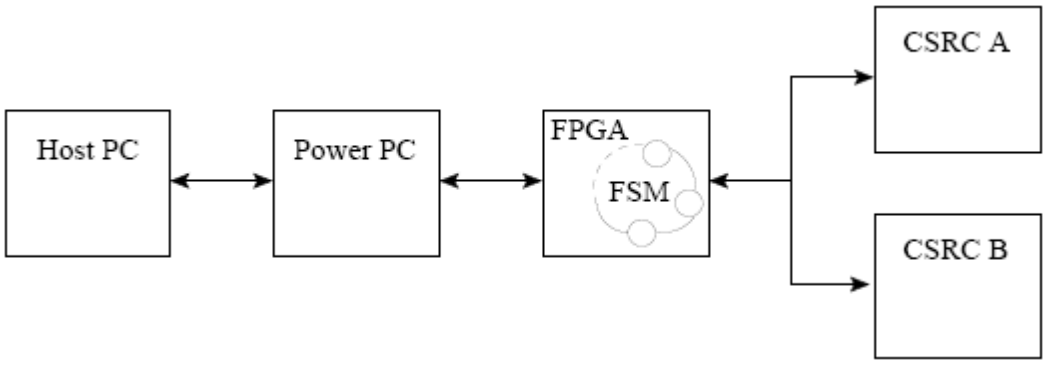
4.3 TRABALHOS MAIS FORTEMENTE RELACIONADOS

Embora na literatura encontre-se ainda poucos relatos de sistemas autonômicos, principalmente na área de hardware, alguns trabalhos fazem menção direta ou indiretamente à essa área. Estes trabalhos foram considerados mais próximos do sistema que propomos na tese, por isso daremos maior ênfase a eles como segue.

4.3.1 Context Switching Strategies in a RunTime Reconfigurable system

As estratégias para mudança do hardware são bastante estudadas na área de Hardware dinâmico, nesse contexto o trabalho apresentado em (PUTTEGOWDA, 2002) mostra uma solução orientada a host de mudar o hardware enquanto uma tarefa está sendo executada.

Figura 4.1 - Arquitetura proposta para a Aplicação Context Switching Strategies in a Runtime Reconfigurable System.



Fonte: (PUTTEGOWDA, 2002)

A arquitetura proposta permite a mudança do hardware atualmente executando de várias formas, no computador host, no processador local do dispositivo ou em uma máquina de estados que está executando dentro do dispositivo reconfigurável. As entidades envolvidas no sistema são mostradas na Figura 4.1, o computador host, o processador local (Power PC), uma máquina de estados (FSM) de controle e os dispositivos de processamento chamados de Context Switch Reconfigurable Cell (CSRC). A aplicação de hardware deve ser implementada usando os CSRC através de arquiteturas propostas por outros trabalhos (SCALERA, 2001). Uma API de acesso ao Power PC é desenvolvida para

permitir a comunicação do Host e a API nativa fornecida pelo fabricante é usada no Power PC para interação com o hardware.

Através da arquitetura proposta os autores podem especificar hardwares reconfiguráveis que podem ser classificados dependendo do nível de reconfiguração utilizado, denominados Orientados à Host quando a reconfiguração é feita no nível do computador host, Orientados à FSM quando a reconfiguração é feita no nível da máquina de estados de controle e Orientados aos Dados onde a reconfiguração é feita em qualquer nível dependendo apenas dos dados tratados e de como eles fluem no hardware.

Para provar o funcionamento da arquitetura os autores implementam três tipos de aplicação dentro de cada um dos tipos de reconfiguração dinâmica suportados fazendo análise de funcionamento e desempenho em cada uma.

Como um estudo generalista em relação à utilização de hardware dinâmico a arquitetura apresentada é bastante completa. O trabalho dos autores implementa com sucesso as aplicações a que foi proposto a implementar mostrando a viabilidade e analisando as vantagens da utilização da arquitetura.

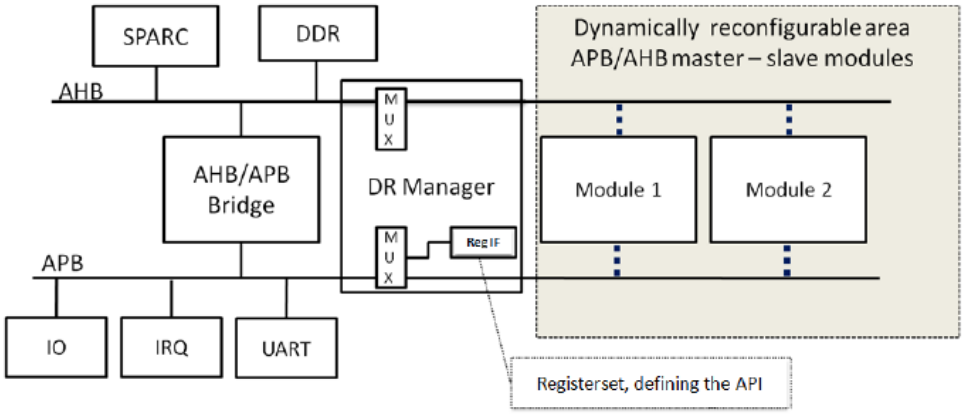
Uma das principais características comuns entre esse trabalho e o apresentado nessa tese é a presença de uma aplicação de controle que deve ser desenvolvida em conjunto do hardware. Essa aplicação, seja ela no nível do host, do Power PC ou da Máquina de Estados será responsável por coordenar qual hardware estará ativo em cada ponto no tempo.

Porém no trabalho o método na geração do hardware que será usado é feito *offline* em termos dos CSRC e gravados no dispositivo reconfigurável. Apenas o controle de qual hardware estará em execução em dado período de tempo é feito pela aplicação. Na nossa arquitetura a geração do hardware é feita em tempo de execução, o que pode demandar mais tempo, porém deixa a solução mais flexível.

4.3.2 The Study of a Dynamic Reconfiguration Manager for Systems-on-Chip

Ainda na área de gerenciamento do hardware e do contexto de memória dos dispositivos este trabalho (KUEHNLE, BRITO, *et al.*, 2011) apresenta uma arquitetura genérica de gerenciamento para hardware em sistemas dinâmicos.

Figura 4.2 - Arquitetura proposta no trabalho DRM.



Fonte: (KUEHNLE, BRITO, *et al.*, 2011)

A Figura 4.2 mostra a arquitetura básica do sistema proposto, bem como componentes de hardware específicos usados para a arquitetura de testes utilizada. O sistema em si começa a partir do componente DR Manager que faz a configuração do hardware em execução. O manager controla um número fixo de regiões reconfiguráveis denominada "*Slots*", tais regiões podem conter no máximo um Módulo que representa um componente de hardware.

O sistema supõe que os *bitstreams* que representam os Módulos são acessíveis de alguma forma ao sistema gerenciador, desse modo o DRM oferece uma API de manipulação que possibilita a gravação de um dado módulo em certo *slot*. O mapeamento das entradas e saídas dos módulos habilitados é feito em memória e o acesso é feito via software diretamente.

O sistema é avaliado através de uma simulação utilizando SystemC (GROTKER, 2002) e uma implementação em hardware Xilinx. São feitas medições espaciais e de potência dos dispositivos que compõem a arquitetura mostrando a viabilidade da mesma em relação à aplicação usada nos experimentos.

O sistema mostrado é fortemente relacionado com o AHM em termos de funcionalidade básica. O sistema AHM prevê o gerenciamento dos componentes de hardware residentes na área reconfigurável do dispositivo, porém ele foca na representação lógica de conexão e comunicação desses componentes não na representação espacial como proposto no DRM.

Em adição nosso sistema prevê que tanto os componentes de hardware quanto a arquitetura formada por sua interligação possa variar com o tempo sendo

controlados pela aplicação em execução, dessa forma a geração *offline* dos *bitstreams* não é viável como a proposta na solução do DRM.

De fato a arquitetura do DRM pode ser utilizada pelo AHM como um gerenciador do espaço ocupado pelo hardware gerenciado ou mesmo executar o escalonamento dos componentes de hardware, se a aplicação necessitar. A arquitetura do DRM não foca na representação da arquitetura de hardware que executam no dispositivo, focando apenas na gerência das regiões reconfiguráveis no decorrer do tempo. Embora o AHM também foque na gerência das regiões reconfiguráveis, ele também proporciona o gerenciamento e geração das arquiteturas de hardware que executam nas regiões gerenciadas.

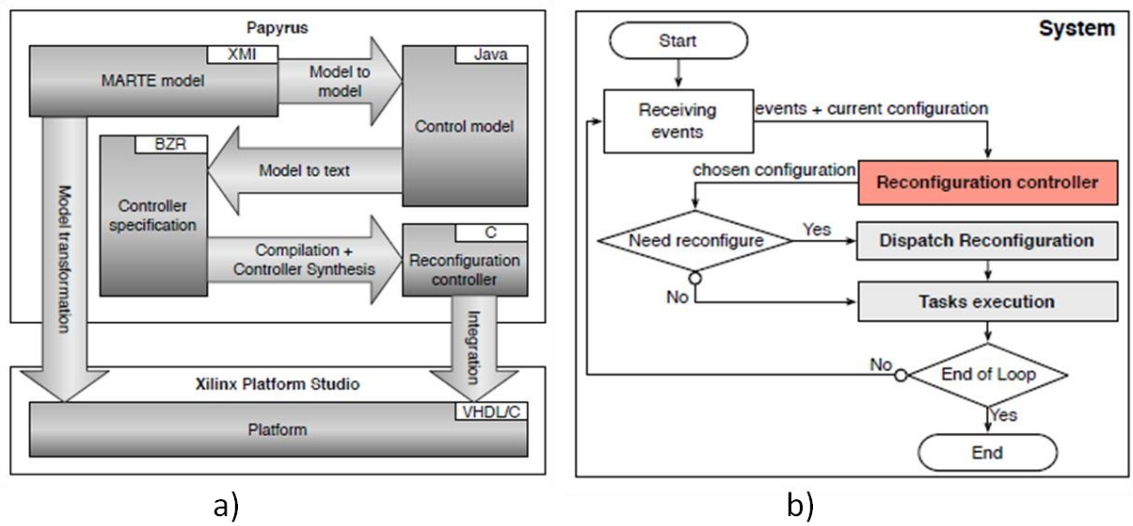
4.3.3 Designing formal reconfiguration control using UML and MARTE

Ainda na área de gerenciamento de regiões reconfiguráveis durante a execução da aplicação (GUILLET, LAMOTTE, *et al.*, 2012), este trabalho define uma metodologia de projeto e execução de aplicações adaptáveis usando o modelo de componentes MARTE e definições em UML para os componentes presentes na aplicação.

Como mostrado na Figura 4.3.a, o esquema propõe mudanças na etapa de projeto onde uma modelagem usando MARTE (OBJECT MANAGEMENT GROUP INC., 2007) é sugerida, nessa modelagem o sistema é descrito em termos de componentes, que possuem modos de operação, a presença ou não de um componente em um dado modo de operação no sistema é descrita por uma máquina de estados controladora.

Um dado conjunto de componentes em certo modo de operação é denominado Configuração. Assim o trabalho do controlador é disparar as transições que modificam os componentes de hardware criando assim novas configurações durante a execução. Os autores propõem que a modelagem tanto dos componentes quanto dos controladores seja feita usando a linguagem MARTE, pois dados os modelos, modos de operação e transições de estado, um modelo na linguagem BZR (DELAVAL, MARCHAND e RUTTEN, 2010) pode ser automaticamente gerado e sintetizado em hardware usando ferramentas disponíveis em um ambiente de projeto denominado Papyrus (LANUSSE, TANGUY, *et al.*, 2009).

Figura 4.3 - a) Projeto do sistema usando a metodologia proposta; b) Execução do sistema usando a metodologia proposta.



Fonte: (GUILLET, LAMOTTE, *et al.*, 2012)

Durante a execução, o sistema usa o diagrama da Figura 4.3.b, que se assemelha bastante com as outras soluções que gerenciam as configurações de dispositivos reconfiguráveis ao decorrer da aplicação. Porém a solução apresentada usando o ambiente Papyrus possibilita a geração automática do sistema de controle de configuração além de propor uma medida de classificação e encapsulamento para os componentes de hardware em suas configurações.

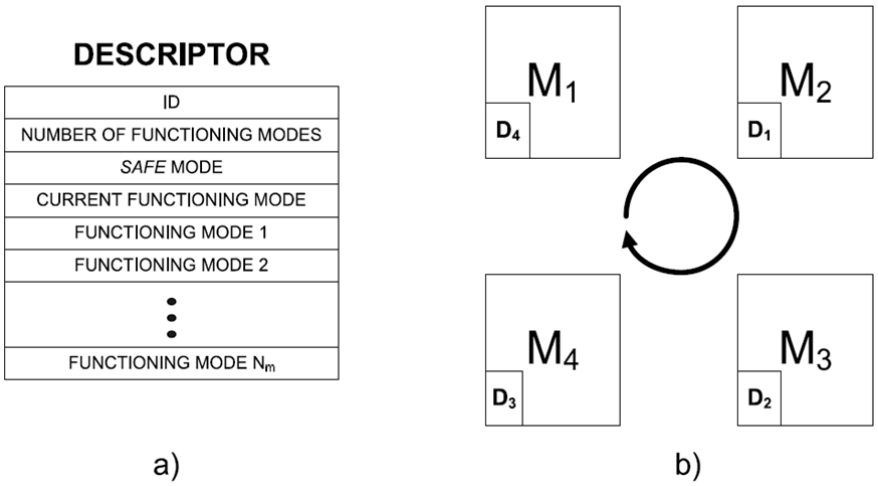
A solução proposta se aproxima com a solução de orientação a contexto proposta pelo AHM, onde as variáveis de contexto importantes para o sistema são definidas durante o projeto e as definições dos eventos que modificam o hardware são descritas através de relações booleanas dessas variáveis. Porém a geração dos componentes e consequentemente das configurações de cada um é feita durante a execução no AHM, enquanto no sistema em questão essa geração é feita de forma *offline*.

O encapsulamento dos componentes através de uma linguagem mais descritiva do que HDL também é outro fato similar entre a solução apresentada nesta Tese e o trabalho em questão. A principal diferença é que para o AHM a descrição XML do componente de hardware é uma descrição mais estrutural focada em ajudar o sistema a construir novas arquiteturas de hardware, enquanto na solução em questão essa descrição é funcional visando descrever o componente em termos de suas funções embora ainda possua uma fraca descrição estrutural.

4.3.4 A New Self-Managing Hardware Design Approach for FPGA-based Reconfigurable Systems

Este trabalho (JOVANOVIC´, TANOUGAST e WEBER, 2008) mostra um sistema capaz de realizar auto-gerenciamento. A arquitetura mostrada se baseia no projeto e implementação de um mecanismo de fluxo de gerenciamento denominado FLUX. A Figura 4.4.b mostra a interconexão entre módulos gerenciados pelo FLUX, os quatro módulos representados trocam informações a respeito de suas funcionalidades, descritas através de uma estrutura que também é ilustrada na Figura 4.4.a.

Figura 4.4 - a) Descrição dos elementos internos do FLUX; b) Diagrama de interconexão entre quatro módulos gerenciados pelo sistema FLUX.



Fonte: (JOVANOVIC´, TANOUGAST e WEBER, 2008)

O principal objetivo desse trabalho é possibilitar o conceito de *awareness* e auto-gerenciamento dos componentes representados pelos módulos M1 à M4. Através do sistema de monitoramento proposto no trabalho o sistema permite que os módulos identifiquem quais as funcionalidades dos seus N vizinhos realizando, durante a execução, o monitoramento a maneira como os vizinhos realizam suas funções. No caso de defeito em algum dos módulos, os vizinhos se adaptam ou tentam encontrar outro módulo dentre os vizinhos que consiga se adaptar para que realize parcial ou totalmente a função do módulo defeituoso.

Para validação da arquitetura, os autores implementam uma aplicação de processamento digital de imagens, simulando falhas nos módulos que executam o processamento. Durante a execução, ao encontrar os módulos defeituosos através

do FLUX, o sistema modifica alguns módulos a fim de se adaptar aos defeitos induzidos durante o experimento.

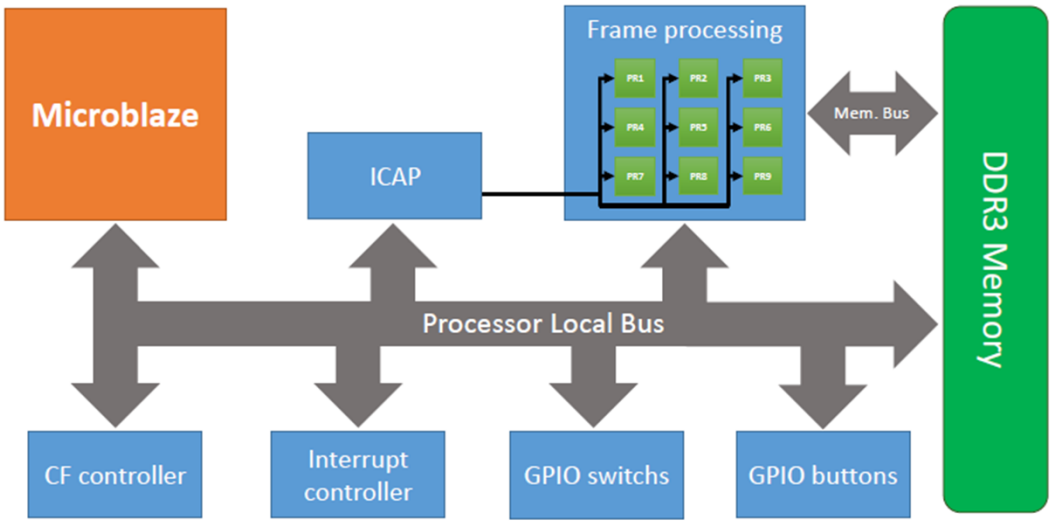
O sistema de auto-gerenciamento mostrado sugere uma solução para o *awareness* dos componentes de hardware, a modelagem da aplicação é feita através dos módulos propostos no trabalho que são interconectados através do FLUX. Embora novos módulos possam ser configurados em tempo de execução e adaptados para realizar novas funções, o trabalho não cita como a entidade gerenciadora poderia gerar novas arquiteturas de hardware, que é um dos focos do AHM. Ainda um ponto a ser mostrado, é que a arquitetura propõe que os módulos estejam pré-instalados na região reconfigurável, significando que eles são projetados, instalados e configurados durante a execução, a arquitetura proposta no AHM possibilitaria à entidade gerenciadora, por exemplo, a relocação dos módulos ou mesmo modificações no projeto dos mesmos de forma automática.

4.3.5 Autonomic Management of Reconfigurable Embedded Systems using Discrete Control: Application to FPGA, 2013

Usando os conceitos da linguagem MARTE e descrição de componentes usando UML os autores implementaram um sistema autonômico de processamento de imagens que se adapta através de uma máquina de estados criada durante o projeto (AN, RUTTEN, *et al.*, 2013).

A Figura 4.5, representa a arquitetura do sistema, é possível perceber que o controlador da região reconfigurável, *CF Controller*, é sintetizado na região estática enquanto os elementos gerenciados estão nas regiões reconfiguráveis PR1 à PR8, representadas no componente *Frame Processing*. Os *bitstreams* referentes às configurações de cada PR ficam alocados em um sistema de armazenamento externo.

Figura 4.5 - Arquitetura do sistema proposto no trabalho, mostrando o componente de controle e a região reconfigurável.



Fonte: (AN, RUTTEN, *et al.*, 2013)

Como descrito anteriormente nas soluções usando MARTE, o controlador é projetado para reagir a eventos, adaptando a arquitetura de hardware. Para o sistema autonômico, os eventos foram retirados a partir de uma modelagem de alto nível dos objetivos do sistema: reduzir o consumo de potência e energia durante a execução; evitar que mais de dois componentes acessem a memória ao mesmo tempo; evitar a sobreposição de tarefas que usam as mesmas regiões configuráveis e garantir a execução completa da tarefa como prioridade máxima.

Esses objetivos são traduzidos em linguagem BZR que, dadas as representações de todos os componentes disponíveis junto com sua qualificação segundo os parâmetros de consumo utilizados, se encarrega de gerar um controlador para o sistema reconfigurável que respeite os parâmetros definidos.

Como o foco do artigo é mostrar a possibilidade de geração dos controladores para aquele conjunto de restrições específicas, os resultados são mostrados em uma tabela que sumariza o tempo levado para criar o controlador dado um conjunto de restrições e um número de possíveis modos de operação para cada um dos componentes.

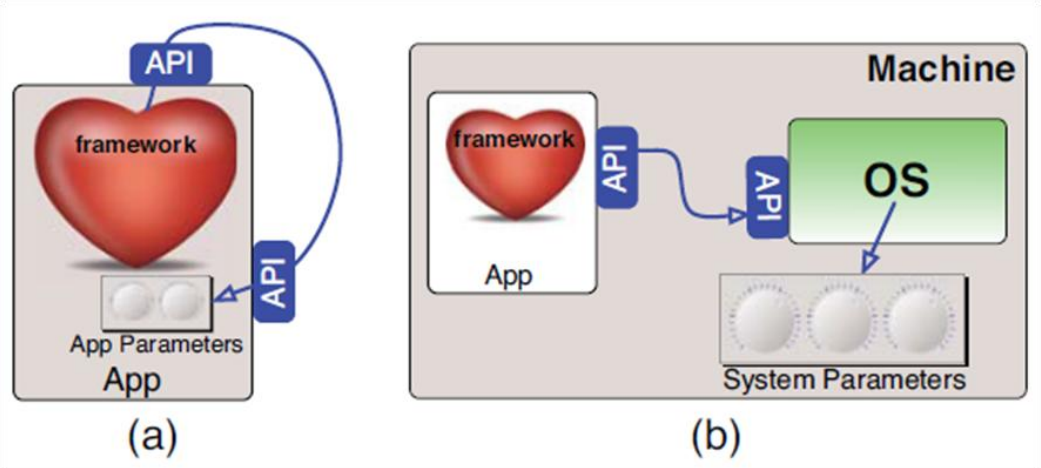
Esse trabalho se relaciona com a arquitetura proposta pelo AHM no que se diz respeito tanto à representação de componentes quanto à definição de variáveis relevantes ao funcionamento. Porém o sistema não faz referência à geração

dinâmica das arquiteturas de hardware, na verdade todo o processo de concepção é feito *offline*, enquanto durante execução os *bitstreams* são acessados e gravados na FPGA.

4.3.6 Framework Application Heartbeats

Este trabalho (HOFFMANN, EASTEP, *et al.*, 2010) apresenta uma solução mais parecida com a proposta desta tese, dividindo o problema de monitoramento e otimização em duas entidades. A solução é baseada no *Framework Application Heartbeats*, que especifica uma API comum para monitoramento e publicação de dados relativos ao desempenho de uma aplicação qualquer.

Figura 4.6 - a) Aplicação monitorada através do *Framework Application Heartbeats*; b) Aplicação que realiza a otimização e monitoramento da performance usando a API proposta no *Application Heartbeats*.



Fonte: (HOFFMANN, EASTEP, *et al.*, 2010)

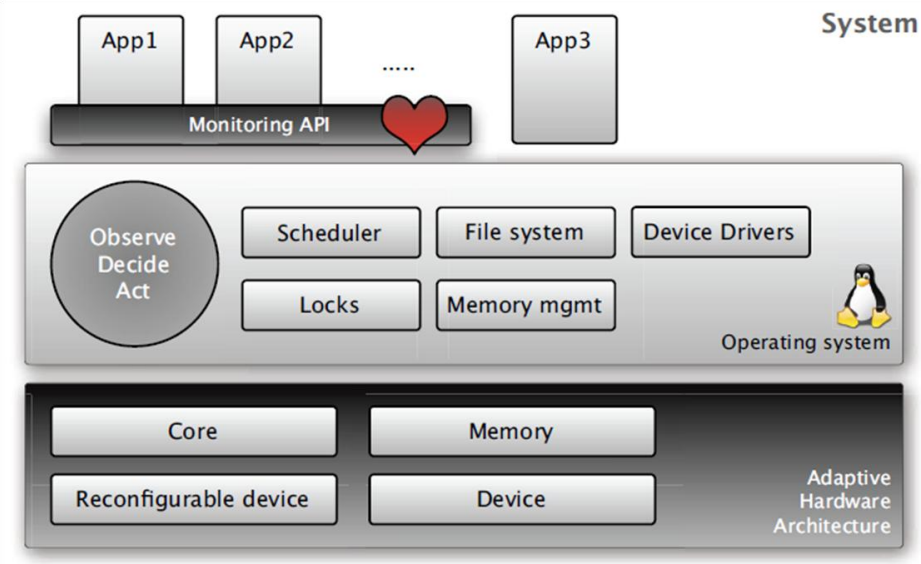
Como mostrado na Figura 4.6, o *Application Heartbeats* monitora e divulga resultados relacionados com o desempenho da aplicação, os resultados são então processados por alguma entidade externa a fim de aperfeiçoar a execução da aplicação monitorada. A entidade externa, comumente, possui maior poder de processamento além de um modelo dos parâmetros importantes para funcionamento do sistema.

O *Application Heartbeats* e as aplicações de teste desenvolvidas pelos autores no decorrer dos anos agrupam algumas das funcionalidades providas pelo AHM em termos de gerenciamento de arquitetura. Por exemplo, quando usado para monitorar sistemas de software ou ajustar a velocidade de processamento da CPU,

a fim de que o sistema se mantenha em certo padrão de consumo de energia (HOFFMANN, MISAILOVIC, *et al.*, 2009) (SANTAMBROGIO, HOFFMANN, *et al.*, 2010).

Seguindo no desenvolvimento usando a API *Heartbeats* os autores propõem um sistema autonômico que, usando o framework, pode escolher automaticamente entre implementações de hardware ou software para uma aplicação ou parte de uma aplicação especifica (SIRONI, TRIVERIO, *et al.*, 2010).

Figura 4.7 - Arquitetura geral usada no sistema autonômico através da API Heartbeats.



Fonte: (SIRONI, TRIVERIO, *et al.*, 2010).

Como mostra a Figura 4.7, a aplicação usa o framework de monitoramento em um sistema embarcado implementado em um dispositivo reconfigurável. Usando as interfaces de monitoramento do framework, o sistema consegue fazer escolhas automáticas entre implementar parte da aplicação em software ou em hardware, caso a performance se degrade devido a fatores externos.

O mecanismo que escolhe entre implementações de hardware e software utiliza carregamento dinâmico de bibliotecas ou reconfiguração parcial para sintetizar o hardware necessário caso o sistema de gerenciamento demande. A mudança entre implementações é feita durante a execução através de um mecanismo de "hot-swap".

Embora o *Application Heartbeats* não abranja o uso de geração automática de hardware, seu mecanismo de monitoramento e otimização se aproxima bastante ao que é proposto no AHM. Porém, devido sua dependência de uma API de

monitoramento e trechos de código que devem ser inseridos na aplicação monitorada, o *Application Heartbeats* não pode ser aplicado para monitorar componentes de hardware. O que é feito na aplicação autonômica descrita é o monitoramento das interfaces de software que acessam o hardware e não do componente de hardware. Em adição, não é mencionado no trabalho se existe algum padrão para os componentes de hardware, presentes no sistema monitorado, ou mesmo como o gerenciamento destes componentes é feito.

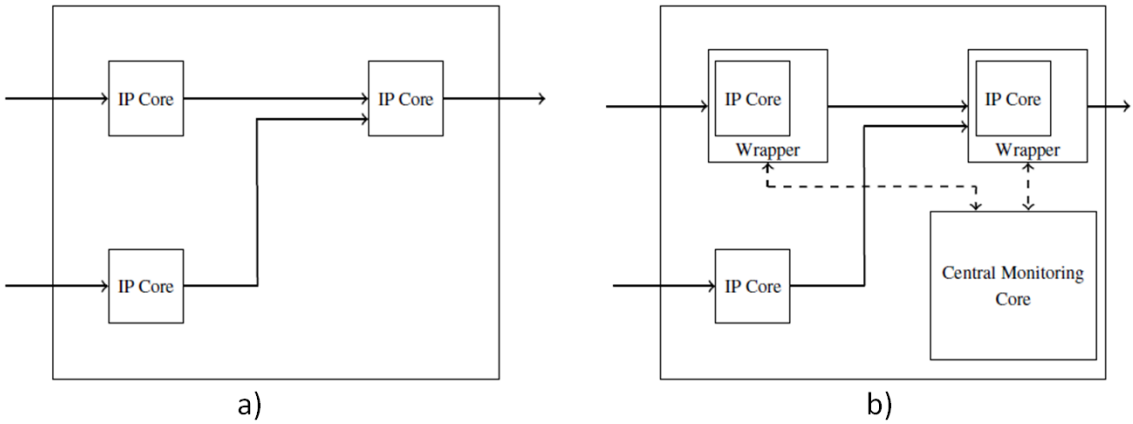
Devido à geração automática da descrição das arquiteturas de hardware, o AHM pode realizar monitoramento tanto de parâmetros de software quanto de hardware. Embora a publicação dos parâmetros observados não seja o foco do AHM, esta é realizada através de um simples protocolo de rede, que envia à entidade gerenciadora uma sequência de chave-valor contendo todos os valores relativos aos parâmetros monitorados, diferente do esquema mais sofisticado proposto pelo *Heartbeats*.

Em adição, o sistema de tomada de decisão proposto pelos autores na aplicação autonômica implementada, foca no desempenho do sistema, que é avaliado através da API de monitoramento. Porém, a implementação não leva em conta outras variáveis externas que possam influenciar o funcionamento. Já no AHM um esquema de monitoramento mais genérico é proposto. Usando pares de chave-valor como variáveis de medida, a aplicação pode decidir o que deve ser levado em consideração e quais ações devem ser tomadas caso hajam variações predeterminadas.

4.3.7 Enhancing FPGA Robustness via generic monitoring IP Cores

Embora não inspirado no esquema de monitoramento do *Application Heartbeats*, este trabalho (BIEDERMANN, PIPER, *et al.*, 2011) apresenta uma solução de monitoramento mais focada em componentes de hardware. No esquema proposto, através de modificações durante a etapa de projeto, componentes de monitoramento podem ser adicionados. Tais componentes observam informações relevantes aos IP Cores monitorados.

Figura 4.8 - a) Sistema de hardware clássico; b) Sistema de hardware usando os componentes de monitoramento descritos.



Fonte: (BIEDERMANN, PIPER, *et al.*, 2011)

O sistema descrito na Figura 4.8 consiste em dois componentes principais, os *Module Monitoring Wrappers* (MMW) e o *Central Monitoring Core*. Os MMW consistem de módulos que espelham as saídas e/ou entradas de um IP Core, podendo checar os valores dessas saídas e entradas através de funções pré-definidas. Um MMW pode, por exemplo, checar se os valores de entrada e/ou saída de uma interface estão restritos a certo conjunto de valores. Uma vez que valores fora desse conjunto são detectados, o MMW pode reagir reportando à entidade de ao *Central Monitoring Core*.

O *Central Monitoring Core* é responsável por monitorar os MMWs. Essa entidade tem o objetivo de reunir informações providas pelos MMW com o objetivo de dar uma visão geral do funcionamento do sistema de hardware.

Nesta Tese propomos no AHM um sistema de monitoramento de componentes de hardware, os Observadores de Interfaces de Hardware, embora o sistema proposto no trabalho em questão seja mais elaborado uma vez que ele pode definir algum nível de filtragem da informação observada nas entradas e saídas dos componentes. Porém os MMW e *Central Monitoring Core* apresentados são adicionados durante o projeto, portanto são estáticos na arquitetura de hardware observada. Já no AHM, os Observadores de Interface são adicionados a medida que a observação de um certo componente de hardware é necessária, dado que as informações de observação são retiradas dos arquivos de descrição do contexto da aplicação. Os autores utilizam, para comunicação do módulo de monitoramento, a interface FLS (ROSINGER, 2004) para realizar a conexão direta com o processador,

enquanto no AHM a interface de comunicação é mais simples usando um protocolo serial permitindo a conexão com qualquer barramento.

4.4 OUTROS TRABALHOS RELACIONADOS

O principal estudo de caso da nossa arquitetura é um sistema autonômico, implementado através de um SBC orientado ao contexto capaz de criar e gerenciar arquiteturas de hardware em execução. A mudança do hardware em tempo de execução não é uma tarefa incomum nas aplicações embarcadas em dispositivos reconfiguráveis.

Efficiently Scheduling Runtime Reconfigurations (RESANO, CLEMENTE, *et al.*, 2008) é um trabalho que leva em consideração a execução de um algoritmo dividido em tarefas que devem ser executadas em determinada ordem, cada tarefa (ou subtarefa) deve ser executada por um hardware diferente que deve ser configurado a cada vez que essa tarefa for ser executada. *Exploiting Partial Runtime Reconfiguration for High-Performance Reconfigurable Computing* (EL-ARABY, GONZALEZ e EL-GHAZAWI, 2009) é um trabalho que utiliza reconfiguração parcial empregada em aumentar a performance de sistemas de otimização, fazendo a modificação do hardware quando a relação entre desempenho e tempo de reconfiguração é boa.

Ambas as soluções utilizam modificação do hardware em tempo de execução visando a melhorias em aplicações. Suas arquiteturas são adaptadas em tempo de execução, mas não tratam da geração de hardware ou mesmo da automação do processo de monitoramento do hardware.

Voltando à área de reconfiguração usando processamento de *bitstream* o trabalho *Supporting Efficient Partial Reconfiguration with Just-In-Time Compilation* (SIDIROPOULOS, SIOZOS, *et al.*, 2012) apresenta uma solução no que se diz respeito à geração de hardware em tempo de execução. Na arquitetura proposta no trabalho os autores utilizam comunicação com um sistema externo que possibilita a geração do hardware automaticamente.

Essa técnica é usada pelo fato de que determinadas arquiteturas de hardware são feitas para ocupar uma área espaça, não obrigatoriamente ocupando todo o espaço disponível. Esse problema é resolvido utilizando uma solução que faz

rastreamento da área atualmente ocupada na FPGA e gera novas arquiteturas de hardware de forma que haja a menor fragmentação possível, deixando assim os componentes mais próximos uns dos outros, melhorando o desempenho geral da Arquitetura de Hardware.

Essa técnica é bastante semelhante a que é usada na arquitetura proposta nesta Tese, supondo que os sistemas dinâmicos possuem requisitos que podem mudar ao longo do tempo os componentes de hardware também devem acompanhar essa mudança, por tanto não podem ser fixos no dispositivo reconfigurável. Para isso é necessário que eles sejam gerados em tempo de execução quando for necessário.

4.5 COMPARAÇÃO ENTRE OS TRABALHOS RELACIONADOS E A TESE

Nesta sessão foram apresentados trabalhos relacionados com a tese proposta, entre a apresentação de um trabalho e do próximo tentamos apontar as principais diferenças entre os trabalhos. Para melhor comparar as soluções listadas, resolvemos classificá-las segundo alguns critérios que estão listados abaixo:

1- Permite reconfiguração de hardware durante a execução.

2- Modelagem de hardware usando técnicas de CBS.

3- Monitoramento de interfaces dos componentes/arquitetura de hardware.

4- Tipo de monitoramento dos componentes distribuído (D) ou local(L).

5- Geração de componentes/arquiteturas de hardware de forma automática durante a execução.

6- Monitoramento de variáveis internas e/ou externas relevantes à aplicação.

Quadro 4.1 - Comparação entre os trabalhos considerados mais próximos do trabalho apresentado nesta tese. “X” significa que o trabalho contém a característica; - significa que o trabalho não implementa a característica; “L/D” refere-se ao tipo de monitoram usado Local(L) ou Distribuído(D), dos componentes gerenciados.

Trabalho/Critério	1	2	3	4	5	6
Context Switching Strategies in a Run-Time Reconfigurable system	X	-	-	L	-	-
The Study of a Dynamic Reconfiguration Manager for Systems-on-Chip	X	-	-	L	-	-
Designing formal reconfiguration control using UML/MARTE	X	X	-	L	X	-

Trabalho/Critério	1	2	3	4	5	6
A New Self-Managing Hardware Design Approach for FPGA-based Reconfigurable Systems	X	-	-	L	-	-
Autonomic Management of Reconfigurable Embedded Systems using Discrete Control: Application to FPGA	X	X	-	L	-	X
Framework Application Heartbeats (Aplicação Autonômica)	X	-	X	D	-	X
Aplicações usando AHM (solução proposta)	X	X	X	D/L	X	X

Fonte: Elaborada pelo autor.

O Quadro 4.1 mostra a comparação entre os trabalhos apresentados seguindo os critérios mencionados. É importante notar que o trabalho *Enhancing FPGA Robustness Via Generic Ip Cores*, também enfatizado nesta sessão, não consta na tabela devido a não se tratar de um sistema de gerenciamento de hardware dinâmico, muito embora o trabalho apresente uma solução próxima à usada pelo AHM no quesito de monitoramento de Componentes de Hardware.

Através de análise do quadro, é possível notar que o AHM se destaca principalmente na capacidade de geração de novas arquiteturas em tempo de execução, além o monitoramento dos Componentes e Arquiteturas de Hardware gerenciados. No quarto quesito, identificamos que o AHM possui um sistema de monitoramento distribuído, mas que pode ser implementado localmente, ignorando as mensagens de monitoramento enviadas pelo *AHM Repository*.

Outro critério que pode ser usado para comparar as soluções encontradas na literatura, são as capacidades autonômicas implementadas, de acordo como (NAMI e BERTELS, 2007), as capacidades autonômicas básicas de um AS podem ser definidas como:

1- Auto-Reparo: Sistemas Autonômicos possuem a habilidade de detectar e reparar falhas a fim de garantir a confiança no funcionamento do sistema.

2- Auto-Otimização: Sistemas autonômicos possuem a habilidade de realizar melhorias automaticamente a fim de manter alcançar os objetivos para qual foram projetados de forma mais eficiente.

3- Auto-Configuração: Sistemas autonômicos possuem habilidade de reflexão para realizar adaptação às variações dinâmicas no ambiente onde estão inseridos.

4- Auto-Proteção: Sistemas autonômicos possuem habilidade de prever e detectar ataques internos e externos, visando garantir a segurança.

Dos quatro atributos básicos mais três podem ser inferidos, como requisitos aos demais (INSAURRALDE, 2013):

5- Auto-descrição: Sistemas autonômicos monitoram o que acontece internamente e externamente ao sistema.

6- Auto-Ajuste: Capacidade de identificar e se ajustar a situações adversas que possam acontecer com o sistema.

7- Auto-Sensibilidade/Sensibilidade ao Contexto: Capacidade do sistema de sentir informações sobre o ambiente onde está inserido.

O Quadro 4.2 compara os sistemas Autonômicos e a solução proposta no AHM, como é possível observar, a capacidade de auto-ajuste em componentes de hardware é suportada apenas por nossa solução uma vez que os outros sistemas trabalham com arquiteturas que são geradas *offline*, inviabilizando a capacidade do sistema se ajustar quando eventos não pré-projetados ocorrem, a menos que todo o projeto seja refeito. Na nossa solução, eventos não projetados inicialmente podem ser inseridos no *AHM Repository* posteriormente, sem necessidade de reprojeto. O servidor ficará em cargo de gerar os novos componentes de hardware e atualizar no *AHM Client* de forma automática realizando assim o auto-ajuste dos componentes.

Quadro 4.2 - Comparação dos sistemas autonômicos estudados em relação as funcionalidades providas. “X” significa que o trabalho contém a funcionalidade; “-“ significa que o trabalho não implementa a funcionalidade.

Trabalho/Critério	1	2	3	4	5	6	7
Designing formal reconfiguration control using UML/MARTE	-	X	X	-	X	-	X
A New Self-Managing Hardware Design Approach for FPGA-based Reconfigurable Systems	X	X	X	-	X	-	-
Autonomic Management of Reconfigurable Embedded Systems using Discrete Control: Application to FPGA	-	X	X	-	X	-	X
Framework Application Heartbeats (Aplicação Autonômica)	-	X	X	-	X	-	-
Aplicações usando AHM (solução proposta)	X	X	X	-	X	X	X

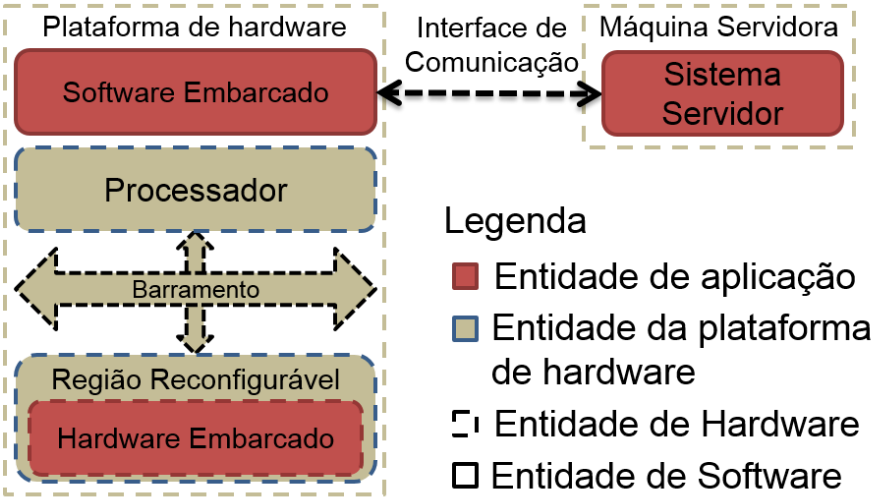
Fonte: Elaborada pelo Autor

Outro ponto a ser observado é que nenhum dos sistemas encontrados na área de sistemas autonômicos de hardware, foca na área de auto-proteção, portanto iremos desconsiderar a comparação neste campo em tópicos futuros.

CAPÍTULO 5
ARQUITETURA PROPOSTA

A fim de situar o leitor a respeito do ambiente em que o sistema proposto nessa tese é executado, a Figura 5.1 exemplifica todos os elementos necessários para que a solução proposta, apresentada nesse capítulo, seja aplicada.

Figura 5.1 - Elementos básicos para aplicação da solução apresentada.

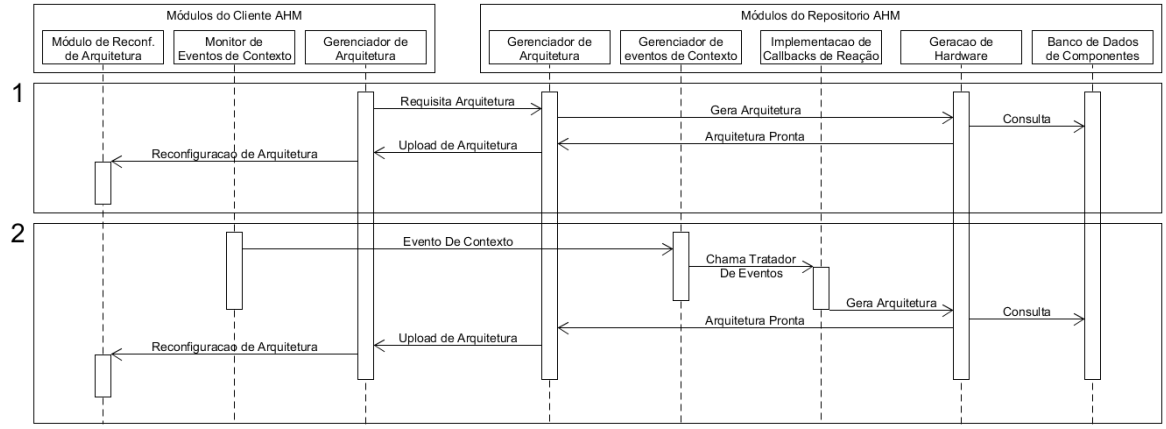


Fonte: Elaborada pelo Autor.

Inicialmente é necessária uma Plataforma de Hardware, com acesso a uma Região Reconfigurável e um Processador que será usado para executar os módulos de software implementados. Um Software Embarcado irá executar sobre o processador provido se comunicando com o Hardware Embarcado através de um Barramento. O processador precisa ser capaz de configurar o hardware presente na Região Reconfigurável através de software. Outro elemento, necessariamente presente, é uma Interface de Comunicação que possibilite a comunicação entre Software Embarcado e o Software do Servidor que executa em uma Máquina Servidora.

Uma vez que esses componentes possuem as características citadas, é possível implementar um sistema que se comporte como gerenciador do hardware presente na região reconfigurável, como será apresentado nesse capítulo.

Figura 5.2 - Diagrama de Sequencia principal: 1) Fluxo básico da geração de arquiteturas de hardware sob demanda. 2) Geração automática de arquiteturas de hardware através de observação de contexto.

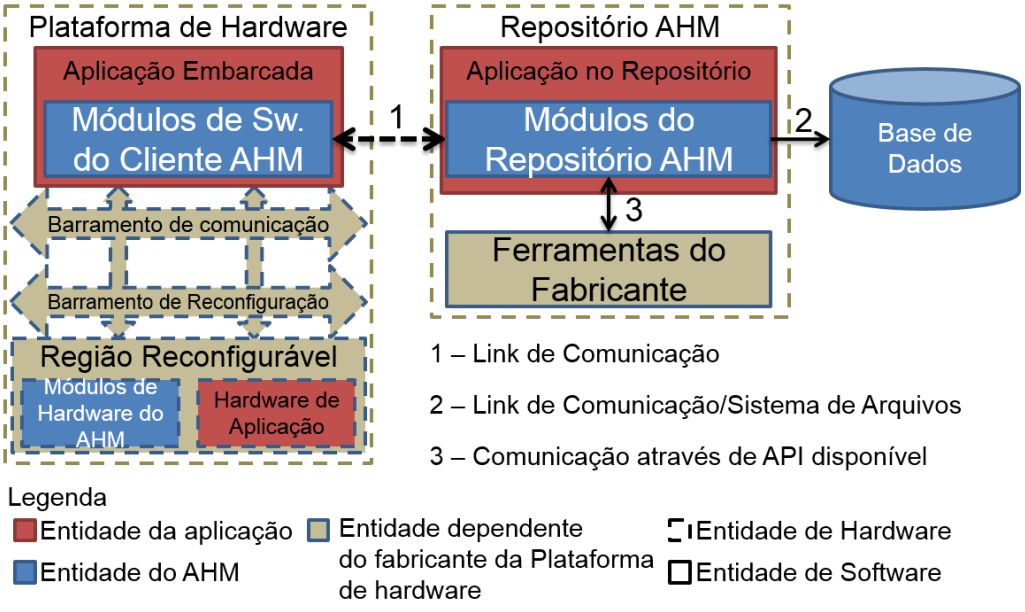


Fonte: Elaborada pelo Autor.

A Figura 5.2 mostra como os dois principais fluxos de execução que o sistema a ser implementado deve suportar, como pode ser visto, no chamado fluxo básico de geração de hardware sob demanda, o hardware embarcado é gerenciado exclusivamente usando os componentes no Cliente AHM e um Repositório de Componentes de Hardware. Adicionando elementos é possível implementar a geração de hardware e gerenciamento automáticos através do monitoramento do contexto do sistema embarcado estendendo o fluxo básico de reconfiguração ilustrado.

Para modelar uma arquitetura que suprisse o diagrama de sequência mostrado, foi desenvolvida uma arquitetura chamada Autonomic Hardware Manager (AHM), usando um repositório ativo de componentes acoplado à aplicação embarcada de forma a prover as características planejadas.

Figura 5.3 - Arquitetura geral proposta para o AHM.



Fonte: Elaborada pelo autor.

A Figura 5.3 mostra a arquitetura geral do sistema cliente-repositório proposto. Para que a arquitetura ficasse mais genérica foram projetadas duas entidades principais, a primeira denominada Cliente AHM e a segunda de Repositório AHM. O tipo de conexão entre as entidades determinará se elas são ou não separadas fisicamente, essa característica foi escolhida devido aos requisitos necessários ao repositório que podem não ser compatíveis com a entidade cliente.

Embora a Figura 5.3 seja executada no mesmo ambiente apresentado anteriormente na Figura 5.1, a fim de simplificar o diagrama, o Processador foi omitido. Porém dentro da entidade Cliente AHM existem elementos de Hardware e Software. Como pode ser visto na figura, parte dos módulos dessa entidade, representados pelo elemento Módulos de Hardware do AHM, são implementados junto ao Hardware da Aplicação dentro da Região Reconfigurável, enquanto o restante é deixado em software os Módulos de Software do Cliente AHM.

Os Módulos do Repositório AHM são responsáveis por se conectar as Bases de Dados tanto de componentes de Hardware quanto de Descrição de Contexto. Os módulos do repositório também se interconectam com as ferramentas

providas pelo fabricante, permitindo assim o processamento das Arquiteturas e Componentes de Hardware.

Dois tipos de barramentos foram descritos visando o caso geral onde o barramento de comunicação é diferente do de reconfiguração, finalmente os três elementos restantes: Aplicação Embarcada; Aplicação do Repositório e Hardware da Aplicação se referem aos elementos que estão fazendo uso das APIs providas pelo AHM bem como sendo gerenciados pelo mesmo.

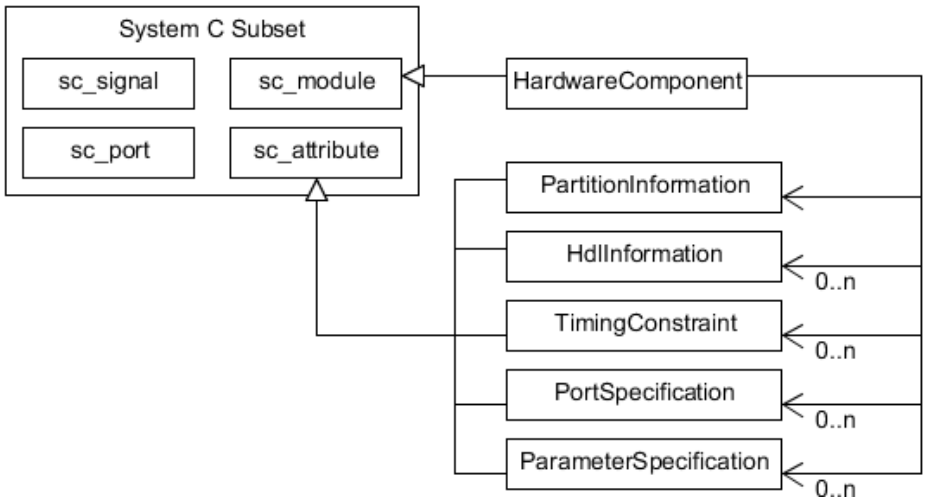
Neste capítulo apresentaremos a arquitetura proposta para a implementação e, posteriormente, usada nos testes do AHM. Inicialmente será apresentada a implementação de um Repositório de Componentes de Hardware, começando pela escolha de um modelo de componentes, seguido pela implementação dos componentes de hardware e software necessários bem como a modelagem do banco de dados de Componentes de Hardware.

Após a implementação do repositório, iremos propor uma extensão de suas características a fim de implementar a sensibilidade ao contexto, requerida para a aplicação do AHM em aplicações autonômicas. Essa extensão será feita primeiramente escolhendo um meta-modelo para o contexto de operação, após isso, novos componentes de hardware e software serão propostos bem como outra modelagem de banco de dados, desta vez para armazenar informações relevantes à descrição do contexto aplicado ao sistema embarcado gerenciado.

5.1 MODELO DE COMPONENTES

Para implementação de um sistema de gerenciamento é necessário escolher, primeiramente, um modelo de componentes. Embora modelos de componentes de software possam ser adaptados para lidar com qualquer tipo de componente, resolvemos escolher um modelo que fosse mais adequado e mais utilizado no contexto de hardware.

Figura 5.4 - Modelo de Componentes escolhido.



Fonte: Elaborada pelo Autor.

Aproveitado a flexibilidade do modelo e sua aplicabilidade para componentes de Hardware, escolhemos utilizar uma versão estendida do modelo de componentes usado no SystemC (GROTKER, 2002). De fato, a API SystemC é conceituada em termos de modelagem, análise e simulação de arquiteturas de hardware. Porém, para nossa necessidade, apenas a parte do modelo de componentes foi utilizada, como mostra a Figura 5.4.

Embora o modelo seja suficiente para uma representação lógica da arquitetura de hardware, o objetivo principal com AHM com esse modelo é permitir a geração de arquiteturas de hardware sob demanda, como já mencionado. Para que a geração seja possível, algumas informações extras precisaram ser adicionadas aos componentes, essa informação é adicionada através das classes implementadas e descritas abaixo:

- *HardwareComponent*: É o tipo principal do modelo de componentes, todos os componentes e arquiteturas gerenciados pelo Repositório e Cliente AHM são instancias (ou aglutinação de instancias) dessa classe.
- *PartitionInformation*: Contém informações acerca da partição reconfigurável para qual o componente que agrega uma instancia dessa classe foi gerado.

- *HdlInformation*: Contém informações a respeito do HDL ou conjunto de HDLs que descrevem o componente que agrega uma instancia dessa classe.
- *TimingConstraint*: Descreve um conjunto de restrições de hardware que devem ser adicionadas durante a geração desse componente.
- *PortSpecification/ParameterSpecification*: Definem os tipos e nomes das portas e parâmetros (*generics* e *parameter*, para vhdl e verilog, respectivamente) de configuração do componente que agrega instâncias dessas classes. Para que seja possível gerar o HDL correspondente a um componente de hardware, informações como tamanho de dados, tipos de sinal (*signed*/*unsigned) e* tipo de dados na interface *(int/float)* são necessárias. Como algumas dessas informações são processadas durante a compilação da entidade em SystemC, essa estrutura possibilita o acesso a esses dados durante a execução.

A interconexão entre portas de componentes é feita através de métodos da classe *HardwareComponent*, ao invés de usar os métodos do SystemC, visando adicionar informação a respeito dos tipos e tamanho dos dados usados, útil à geração de hardware, no processo de processamento de arquiteturas. Com exceção da interconexão entre as portas dos componentes o restante da manipulação dos componentes é feito da mesma forma como no SystemC.

5.2 CLIENTE AHM

O Cliente AHM é uma entidade que deve ser implementada no dispositivo reconfigurável. Como mostrado na Figura 5.3, esse sistema é projetado para interagir com o dispositivo reconfigurável através de dois canais de comunicação: os barramentos de comunicação e de configuração.

O barramento de configuração é um barramento relacionado diretamente com o tipo de hardware reconfigurável e o fabricante do mesmo. A intenção de inserir esse elemento na arquitetura é prever que haverá uma interface definida para configurar o dispositivo. Embora sua implementação e especificação esteja

fora do escopo da tese ele é representado, pois é um elemento crucial para o funcionamento geral da arquitetura.

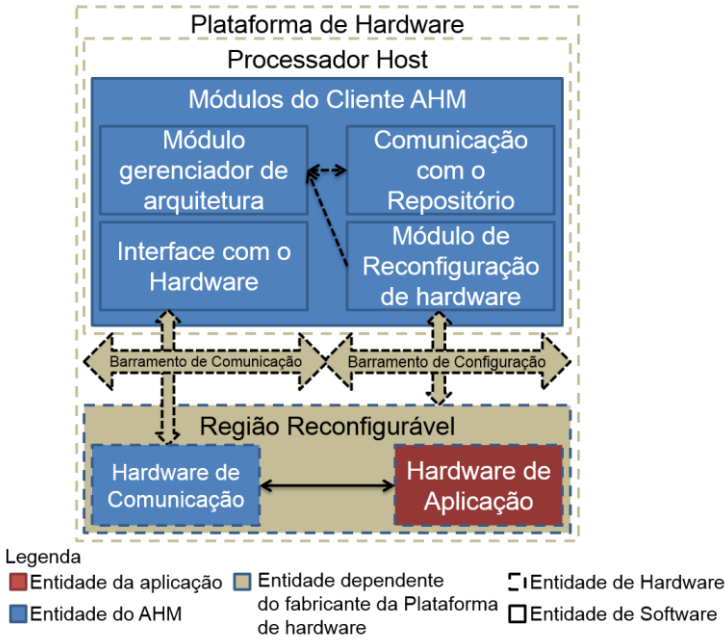
O Barramento de Comunicação possui características mais flexíveis, pois ele é usado para comunicação entre os componentes de hardware gerenciados e as entidades de software providas pelas APIs nos módulos do Cliente AHM. Assim o Barramento de Comunicação pode ser adaptado dependendo das opções de comunicação disponíveis no dispositivo reconfigurável. Esse componente se faz necessário, pois o protocolo de comunicação entre componentes de hardware e software não é bem definido entre os fornecedores de sistemas reconfiguráveis, portanto o Barramento de Comunicação provê um canal bem definido de interface entre o hardware em operação e o sistema de gerenciamento do cliente.

A Figura 5.5 ilustra os cinco módulos, quatro de software e um de hardware, que compõem o sistema cliente. As setas representam a presença de comunicação entre os módulos:

- *Módulo Gerenciador de Arquitetura:* Esse módulo encapsula um sistema de gerenciamento de componentes, ele contém uma representação lógica da arquitetura de componentes, usando o subconjunto do SystemC adotado como modelo de componentes. Ele também é responsável por processar novas arquiteturas recebidas pelo Módulo de Comunicação com o Servidor e adicioná-las no sistema através do Módulo de Configuração do Hardware.
- *Módulo de Comunicação com o Servidor:* Responsável por possibilitar a comunicação com o Repositório de forma centralizada. Ele é responsável por se conectar com o Repositório através de um link de comunicação remoto ou local.
- *Módulo de Configuração do Hardware:* O sistema de configuração do hardware é totalmente dependente do Barramento de Configuração. Por isso esse módulo é responsável por encapsular a implementação dependente de plataforma que será necessária para realizar a configuração do dispositivo.

- *Módulo de Interface com o Hardware:* O sistema de comunicação entre Arquiteturas de hardware e o Software Embarcado não é padronizado. Usando o fluxo de desenvolvimento provido pelo fabricante, a quantidade de saídas e entradas de uma arquitetura de hardware, precisa ser definida durante o projeto. Dessa forma não é possível adicionar ou remover componentes de hardware se essa adição mudasse a quantidade total de interfaces definidas na região reconfigurável durante a etapa de projeto. Para contornar essa limitação, esse módulo foi projetado para permitir um acesso mais genérico aos componentes presentes em uma Arquitetura de Hardware através de APIs de software e interação com o Hardware de Comunicação.
- *Hardware de Comunicação:* Permitir a comunicação com os componentes de hardware que compõem a arquitetura de forma padronizada. O Módulo de Interface com o Hardware utiliza um protocolo de comunicação, que, junto com este módulo, remove a limitação com respeito ao número de entradas e saídas para o hardware que executa na região reconfigurável.

Figura 5.5 - Módulos do Cliente AHM.



Fonte: Elaborada pelo Autor.

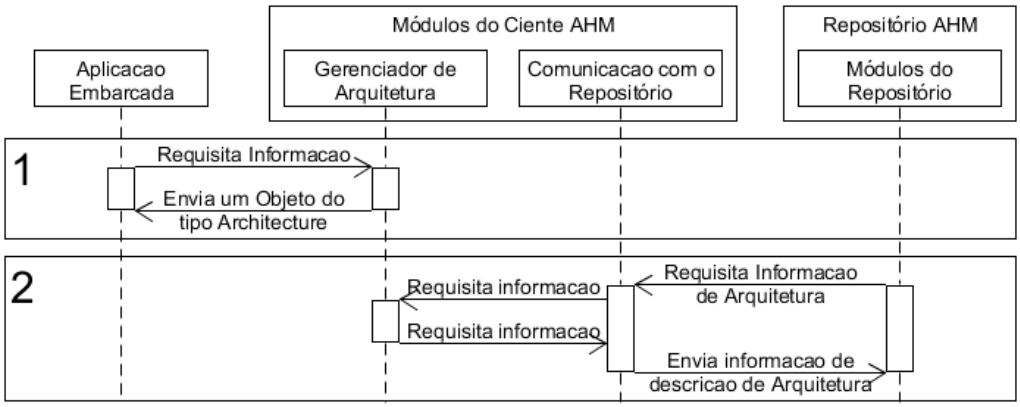
Como já descrito, a descrição dos componentes o Cliente AHM é um composto entre componentes de hardware e componentes de software. Ele pode ser implementado dentro do dispositivo reconfigurável desde que haja uma interface de reconfiguração disponível e que seja possível implementar o Barramento de Comunicação. A arquitetura proposta foi projetada para um sistema que possua uma parte reconfigurável e um processador host que possa ser usado para implementação das interfaces de software. Além disso, é necessário que haja espaço na parte reconfigurável para inserção dos módulos do Hardware de Comunicação.

5.3 FUNCIONALIDADES PROVIDAS PELO CLIENTE AHM

Os módulos do Cliente AHM focam em prover à Aplicação Embarcada funções de gerenciamento e acesso aos Componentes e Arquiteturas de hardware que são gerenciados pelo AHM. Outro objetivo desses módulos é prover métodos de comunicação com o Repositório AHM através da interface de comunicação.

As principais funcionalidades que serão descritas são: Monitoramento da Arquitetura de Hardware presente na região reconfigurável; Suporte ao acesso às interfaces presentes nos Componentes de Hardware gerenciados; Suporte à configuração de Arquiteturas de Hardware nas regiões reconfiguráveis gerenciadas; Comunicação com o Repositório através da interface de comunicação provida; Mecanismo de requisição e download de Arquiteturas de Hardware através de comunicação com o servidor;

Figura 5.6 - Diagrama de sequência para operações de monitoramento da Arquitetura. 1) Monitoramento por meio da Aplicação Embarcada; 2) Monitoramento pelo Repositório AHM.



Fonte: Elaborada pelo Autor.

A Figura 5.6 mostra como são feitas as operações de monitoramento e inspeção da Arquitetura de Hardware gerenciada através dos módulos do Cliente AHM. Em ambos os tipos de operação, o objetivo é obter informações sobre os componentes internos de uma Arquitetura de Hardware, essa informação pode ser tanto a respeito da interconexão entre os Componentes de Hardware, os valores em algumas interfaces dos componentes ou mesmo parâmetros de configuração dos mesmos.

As informações a respeito da interconexão dos Componentes de Hardware além de seus parâmetros de configuração são listadas nos Arquivos de Descrição de Arquitetura. Os módulos ilustrados na Figura 5.6 acessam esses arquivos para criação de suas estruturas de dados internas além de usá-los para prover informações a respeito de outras arquiteturas que possam estar armazenadas, mas ainda não em uso.

Figura 5.7 - a) Estrutura do arquivo de descrição de arquiteturas contidas no cliente AHM; b) Estrutura do arquivo de descrição da configuração padrão para o sistema.

```
architecture <uid> <archName> <regionName>
      communicationTable
            (<portName> <index>)*
      endCommunicationTable
      {component <uniqueCompName>
            (port <portName> <portType>)*
            (param <paramName> <paramType> <defaultValue>)*
       endComponent <uniqueCompName>}*
      {bind <componentAName> <componentAPort> <componentBName>
             <componentBPort>}*
endArchitecture
onde
<uid> => identificador para a arquitetura gerado no repositório.
<archName> => nome da arquitetura.
<regionName> => nome da regiao reconfiguravel para qual essa
                arquitetura foi gerada.
<uniqueCompName> => nome da instancia desse componente dentro da
                    arquitetura corrente
<portName> => nome da porta
<index> => indice gerado pelo Repoisitório para a interface
           Genérica correspondente a uma porta.
<portType> => pertencente ao conjunto: {"in, out, inout"} x
              {"bit", "vector(size ... 0)"}
<paramName> => nome do paramentro para o componente corrente
<paramType> => pertencente ao conjunto : {"string", "integer"}
<attrDefaultValue> => valor padrao para esse parametro
```

a)

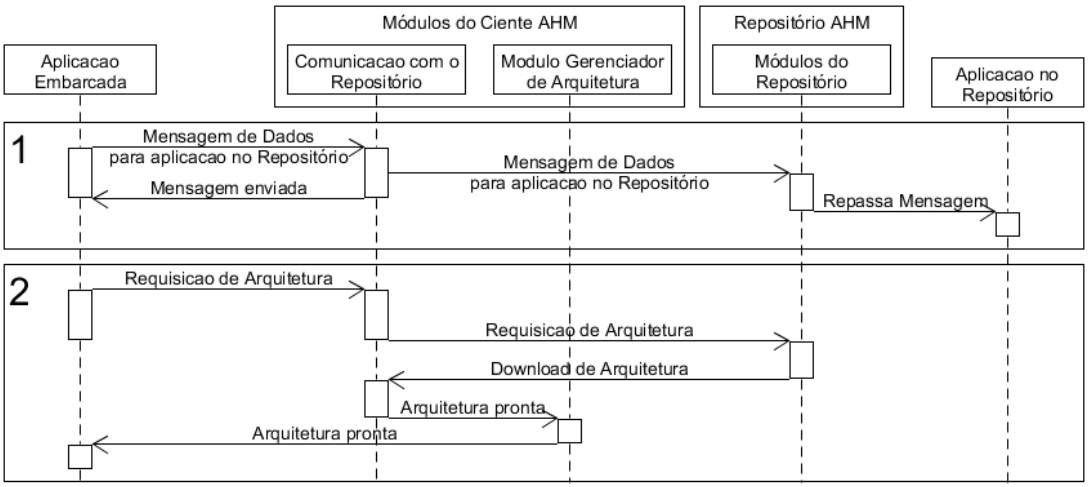
```
{reconfigurableRegion <regionName>
      busWrapperBaseAddress <address>
      architecture <uid>
endReconfigurableRegion} *
onde
<address> = endereco base, no barramento,
            do Hardware de Comunicacao
<regionName> = nome da regiao
<uid> = id gerado no repositório AHM
```

b)

Fonte: Elaborada pelo Autor.

O formato dos arquivos de descrição de arquiteturas contidas no Cliente AHM é ilustrado na Figura 5.7.a, tais arquivos devem ser armazenados a medida que novas arquiteturas são recebidas. Já o formato dos arquivos de descrição de arquitetura padrão é mostrado na Figura 5.7.b, tais arquivos devem ser gerados inicialmente, bem como as arquiteturas referenciadas, durante o projeto e implantação da Aplicação Embarcada.

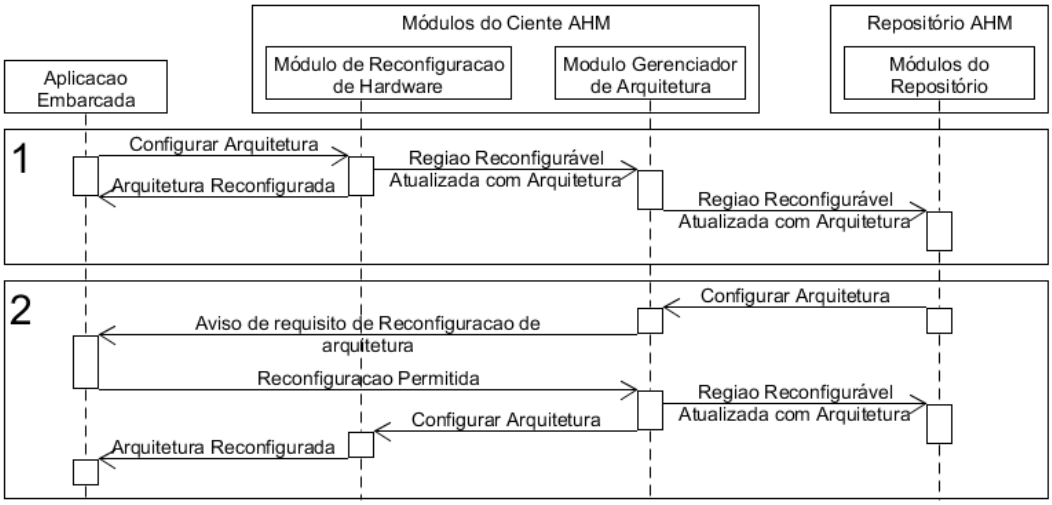
Figura 5.8 - Diagramas de sequência para comunicação usando os módulos do cliente AHM. 1) Comunicação entre a Aplicação Embarcada e a Aplicação no Repositório usando as APIs providas; 2) Comunicação entre a Aplicação Embarcada e o Repositório AHM, requisitando uma Arquitetura de Hardware.



Fonte: Elaborada pelo Autor.

A Figura 5.8 representa dois fluxos básicos de comunicação providos pelos módulos do Cliente AHM. Usando as APIs providas é possível trocar mensagens entre as aplicações Embarcada e no Repositório, além de possibilitar comunicação direta com o repositório AHM, para, por exemplo, requisitar uma Arquitetura de Hardware ou realizar uma consulta no Banco de Dados de Componentes.

Figura 5.9 - Diagramas de sequência para reconfiguração usando os módulos do cliente AHM. 1) Reconfiguração oriunda da Aplicação Embarcada. 2) Reconfiguração oriunda do Repositório AHM.



Fonte: Elaborada pelo Autor.

Finalmente a Figura 5.9 mostra o fluxo de informação nos processos de reconfiguração usando as APIs providas pelo Repositório AHM. É possível notar pela figura que existem dois fluxos básicos que podem originar reconfiguração, um partindo da Aplicação Embarcada através de uma chamada expressa das funções de reconfiguração. Já o outro, parte do Repositório AHM devido às interações com a Aplicação no Repositório.

Ambas as formas de reconfiguração requerem que o *bitstream* da Arquitetura de Hardware a ser configurada esteja disponível no sistema de arquivos quando as APIs do módulo de Reconfiguração de Hardware forem chamadas. Para tanto os Módulos de Comunicação são usados para realizar o *download* dos arquivos de descrição e *bitstreams* das arquiteturas.

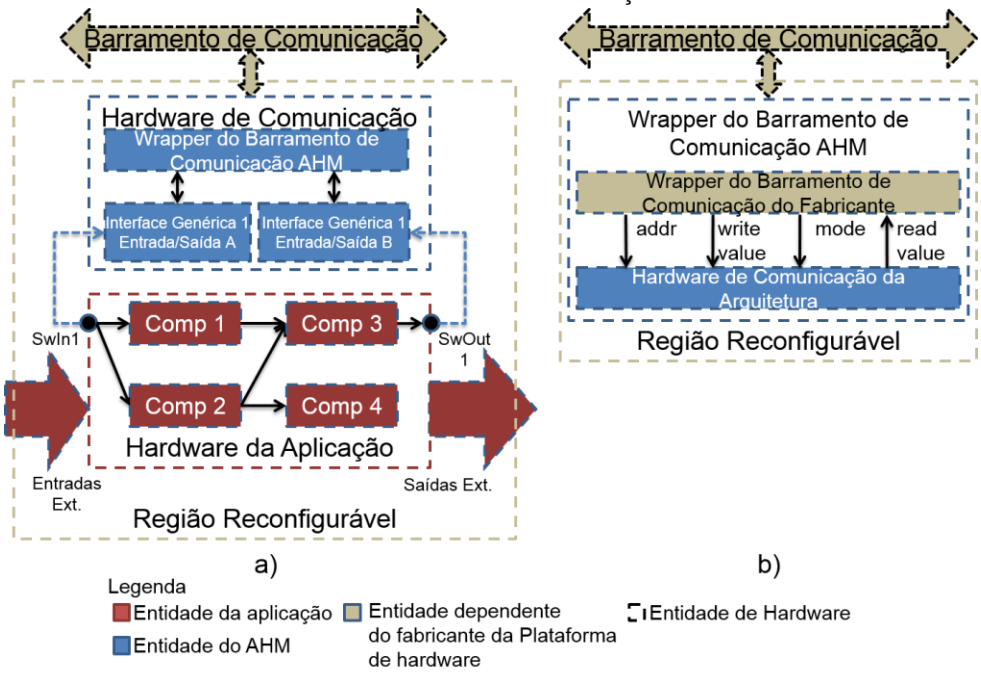
5.4 SISTEMA DE COMUNICAÇÃO COM O HARDWARE NO CLIENTE AHM

Na arquitetura do Cliente AHM, ilustrada na Figura 5.5, é mostrado o módulo Hardware de Comunicação que é usado para implementar a comunicação entre os módulos de software do AHM e o Hardware da Aplicação. Um detalhamento desse módulo é mostrado na

Figura 5.10, ilustrando um sistema de hardware composto de três componentes, cujas saídas dos componentes *Comp 1* e *Comp 3* são observadas pelo Hardware de Comunicação.

A arquitetura do Hardware de Comunicação é composta pelo *Wrapper* do Barramento de Comunicação, o Hardware de Comunicação da Arquitetura e as Interfaces Genéricas. O *Wrapper* do Barramento de Comunicação implementa componente escravo do Barramento de Comunicação disponível e é um componente que precisa ser adicionado durante a etapa de projeto do sistema. Nas implementações realizadas esse componente é gerado automaticamente pelas ferramentas do fabricante, a partir daí framework fornecido pelo AHM gera um esqueleto do componente Hardware de Comunicação da Arquitetura através das especificações da região reconfigurável, e o projetista é responsável por adicionar esse componente ao sistema.

Figura 5.10 - a) Interfaces genéricas geradas no Hardware de Comunicação; b) Wrapper do Barramento de Comunicação



Fonte: Elaborada pelo Autor.

As Interfaces Genéricas, como visto na Figura 5.10.a, são exemplos de interfaces dos Componentes de Hardware de uma Arquitetura que precisem ser acessados pelo software embarcado gerenciado pelo AHM. A quantidade de Interfaces Genéricas não é fixa, nem mesmo seu tamanho de dados, eles são gerados automaticamente dependendo de quantos elementos precisem ser acessados.

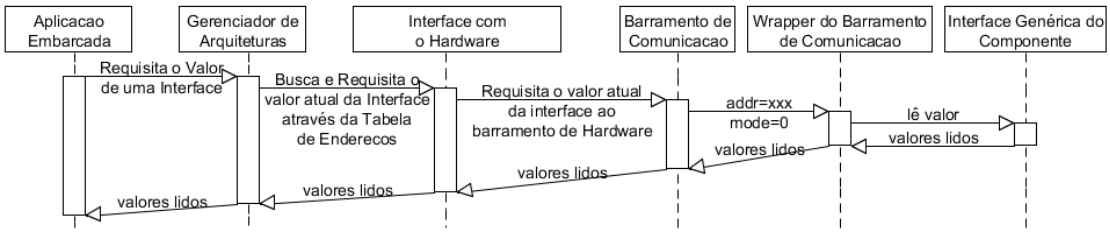
O Hardware de Comunicação da Arquitetura recebe quatro principais entradas, como mostrado na Figura 5.10.b, essas entradas são usadas para endereçar os pontos de acesso às interfaces providas pela arquitetura atualmente configurada na região. Atuando junto com o componente *Interface com o Hardware* esse esquema permite que a arquitetura, atualmente configurada, possua qualquer quantidade de entradas e saídas necessárias, bem como tamanhos variáveis para essas. A leitura e escrita nesses componentes são feitas de forma serial, portanto apenas uma das interfaces pode ser escrita ou lida a cada ciclo.

A Figura 5.11, mostra um diagrama de sequência que exemplifica a comunicação usando a solução proposta, o exemplo pode ser extrapolado para a escrita, mudando apenas o valor do registrador *mode*. Não existe sincronização

automática, a não ser o próprio *clock* do dispositivo reconfigurável, nas operações de leitura ou escrita.

A solução é implementada de modo que os valores das interfaces requisitadas são amostrados em paralelo à execução da aplicação, de forma que as requisições de leitura podem ser feitas de forma assíncrona. Já na escrita os valores são postos em registradores dentro do hardware, portanto é recomendado que um sistema de sincronia seja implementado, no caso em que seja necessário escrever em registradores cujo espaço de armazenamento seja maior que 32 bits.

Figura 5.11 - Diagrama de exemplo da comunicação entre as interfaces de software e hardware, usando o esquema proposto.



Fonte: Elaborada pelo Autor.

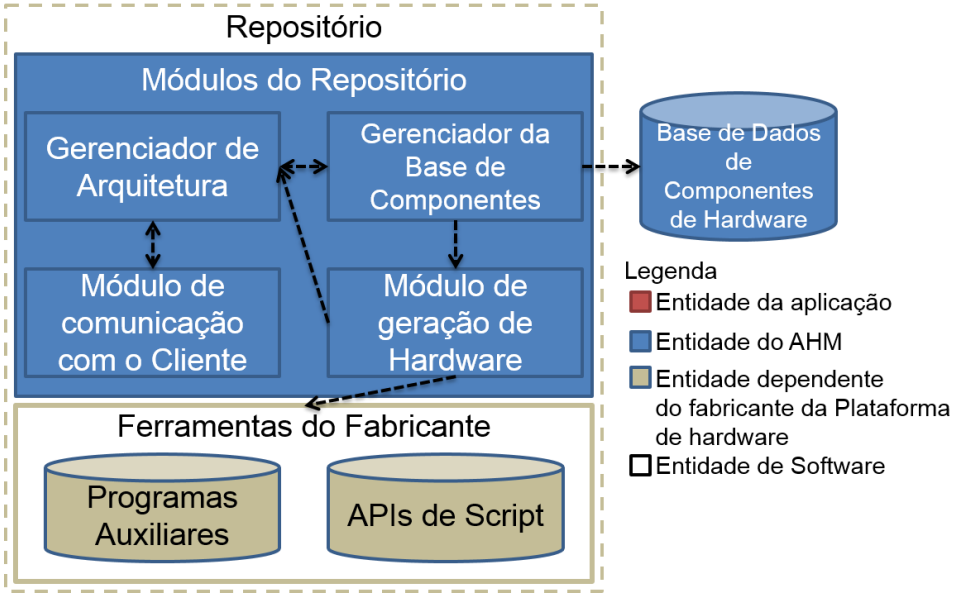
A arquitetura prevê ainda a presença de Entradas e Saídas externas no sistema de hardware devido ao fato de que o Cliente AHM pode não ser capaz de fornecer todas as entradas necessárias para interconectar componentes externos como barramentos ou interfaces específicas. Dessa forma essas entradas e saídas precisam ser definidas e conectadas durante a etapa de projeto e, portanto, não serão passíveis de reconfiguração.

5.5 REPOSITÓRIO AHM

A implementação do sistema repositório cobrirá o acesso à base de dados de componentes e o acesso às ferramentas de geração e manipulação de hardware que são providas pelos fabricantes de hardware reconfigurável. O repositório será, inicialmente, um elemento passivo, gerando arquiteturas quando requisitado para isso precisará prover APIs e métodos de comunicação para possibilitar que as ordens para gerar novas arquiteturas possam vir tanto de um

operador humano, de uma aplicação residente na entidade que executa o Repositório ou do Cliente AHM.

Figura 5.12 - Componentes do Repositório AHM.



Fonte: Elaborada pelo Autor.

Para suportar essa característica o repositório foi projetado como ilustrado na Figura 5.12. Diferente dos componentes de Software os componentes de hardware não possuem uma representação compilada independente de plataforma como as bibliotecas. Para poder tratar os componentes de hardware como componentes de software nesse aspecto o repositório precisa ter acesso direto às descrições em HDL, ou equivalente, dos componentes de hardware. Além disso, o sistema precisa usar algumas ferramentas de compilação para transformar o código em HDL em hardware propriamente dito. Para suprir esses requisitos os componentes do repositório são:

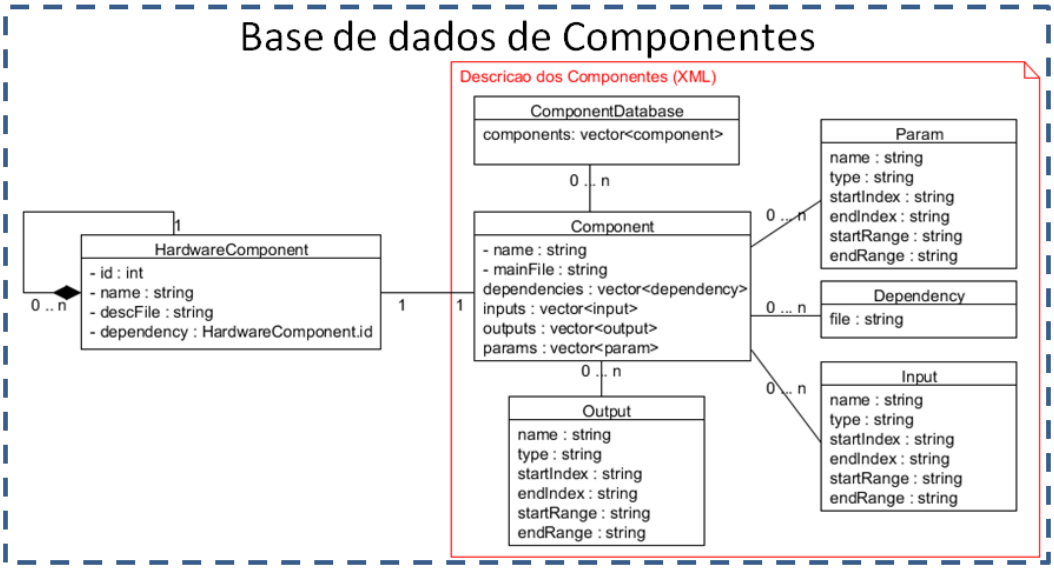
- *Módulo Gerenciador de Arquitetura*: O Repositório AHM possui uma representação lógica da arquitetura de componentes que está atualmente executando no Cliente AHM, utilizando o modelo de componentes adotado. Nesse módulo são mantidas representações da estrutura de componentes para cada Cliente AHM gerenciado, dessa forma cada vez que uma nova arquitetura é requisitada é possível saber

a diferença entre a arquitetura antiga e a que será gerada, informação que é útil para alguns processos de geração de hardware.

- *Módulo de Localização de Hardware*: Dada a representação arquitetural, cada componente possui uma representação em HDL que deve ser acessível ao Servidor. A localização dos arquivos HDL junto com possíveis dependências é gerenciada por esse módulo que pode suportar acesso aos arquivos físicos de forma remota ou local. Haja vista que uma descrição de hardware pode possuir muitas dependências diretas e indiretas, esse módulo usa uma representação em XML, representada na Figura 5.13, que descreve o componente de hardware e lista suas dependências.
- *Módulo de Comunicação com o Cliente*: Esse módulo foi projetado para centralizar as operações de comunicação de forma que o servidor possa ser implementado tanto com conexão remota quanto por chamada de API diretamente do Cliente. Além disso, esse módulo tem as funções de receber requisições e mandar novas arquiteturas para o cliente.
- *Módulo de Geração de Hardware*: A principal função desse módulo é encapsular o uso das APIs e funções dependentes de fabricante. Uma vez gerada uma nova arquitetura o Gerenciador de Arquitetura irá evocar esse módulo fornecendo o HDL gerado junto com todas as dependências recuperadas através do Módulo de Localização de Hardware para realizar a compilação da arquitetura e geração do *bitstream* relativo à mesma.

O repositório acessa os componentes dentro de uma base de dados que são armazenados e acessados segundo uma descrição XML, os atributos presentes na descrição desses componentes bem como a relação entre eles são representados na Figura 5.13.

Figura 5.13 - Diagrama UML representando a estrutura do XML de descrição dos componentes no banco de dados.



Fonte: Elaborada pelo Autor.

Como ilustrado, cada instancia da classe *HardwareComponent* referência um elemento no banco de dados representado pela *tag Component*, esse elemento possui alguns atributos a respeito da entidade descrita além de uma lista de arquivos HDL com a descrição do componente de hardware em questão. Para evitar o processamento do HDL principal, as entradas, saídas e atributos de configuração do componente também precisam ser descritas.

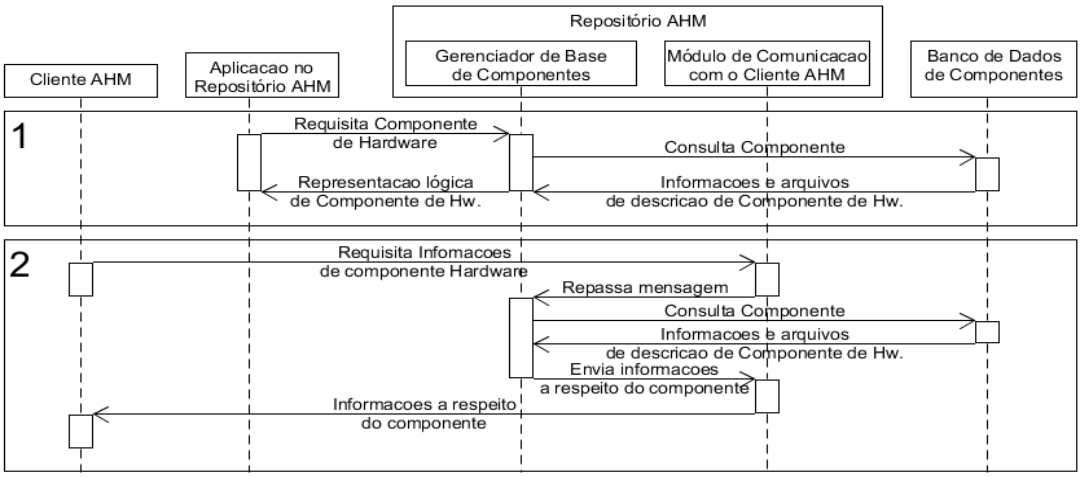
5.6 FUNCIONALIDADES PROVIDAS PELO REPOSITÓRIO AHM

Seguindo o modelo arquitetural apresentado anteriormente, o Repositório AHM implementa as principais funcionalidades de um repositório de componentes de software além de prover mecanismos para que possa ser acessado diretamente por uma eventual aplicação adicional de gerenciamento. Dentre as funcionalidades do Repositório AHM, estão ainda a construção de Arquiteturas de Hardware sob demanda e a criação automática de Componentes de Hardware relativos à comunicação e aspectos estruturais das arquiteturas requisitadas.

Desse modo as principais funcionalidades providas pelo Repositório AHM são: Localizar informação e arquivos relativos a Componentes de Hardware

presentes no *Banco de Dados de Componentes*; Gerar o *bitstream* e arquivos de configuração de Arquiteturas de Hardware através da costura de Componentes de Hardware usando código HDL gerado automaticamente; Criar automaticamente interfaces de acesso, *Tabelas de Comunicação* relativas ao *Hardware de Comunicação* específico para cada Arquitetura de Hardware criada.

Figura 5.14 - Diagrama de sequência mostrando o fluxo dados da funcionalidade de consulta a Base de Dados de Componentes. 1) Consulta partindo do próprio repositório; 2) Consulta partindo do Cliente AHM.

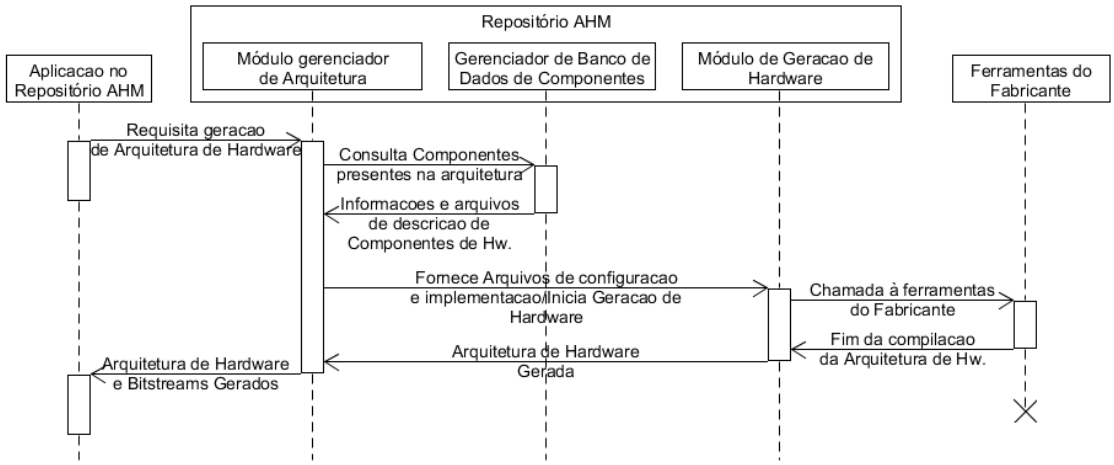


Fonte: Elaborada pelo Autor.

A Figura 5.14, apresenta o fluxo de informações para a funcionalidade de localização e consulta dos Componentes de Hardware presentes no Banco de Dados de Componentes. A figura mostra dois tipos de interação, no primeiro tipo a operação de consulta começa dentro do próprio Repositório. As requisições são passadas diretamente ao Gerenciador de Base de Componentes. Esse componente faz a consulta no Banco de Dados e retornar para a Aplicação no Repositório as informações e uma representação lógica do Componente de Hardware requisitado.

O segundo tipo de operação de consulta, é iniciado no Cliente AHM, nesse caso o cliente envia uma mensagem requisitando informações sobre um componente específico, a mensagem é receptada pelo Módulo de Comunicação com o Cliente AHM, que repassa a requisição para o Gerenciador de Base de Componentes, que, por sua vez, faz a consulta e envia as informações requisitadas de volta ao Cliente AHM.

Figura 5.15 - Diagrama de sequência mostrando o fluxo dados da funcionalidade de consulta a geração de arquivos e *bitstreams* para Arquiteturas de Hardware.

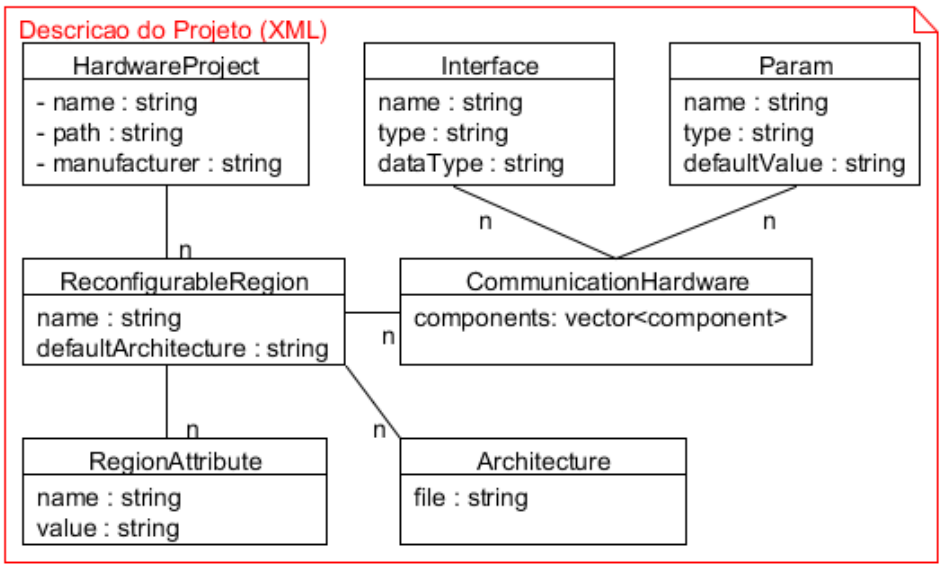


Fonte: Elaborada pelo Autor.

A Figura 5.15 mostra como o Repositório AHM se comporta ao receber uma requisição para gerar o *bitstream* correspondente a uma Arquitetura de Hardware. Embora não seja descrito diretamente na figura, a requisição pode vir tanto da Aplicação no Repositório AHM, quanto remotamente do Cliente AHM, através das APIs de comunicação com o Repositório.

Inicialmente uma arquitetura precisa ser construída usando o Modelo de Componentes adotado e Componentes de Hardware presentes na Base de Dados. Em seguida, o Módulo Gerenciador de Arquitetura pode criar uma estrutura que representa a Arquitetura de Hardware associada a uma Região Reconfigurável. A partir dessa estrutura, a geração do *bitstream* parcial é executada. Ao fim do procedimento são obtidos os arquivos de descrição e os *bitstreams* relativos à Arquitetura de Hardware processada.

Figura 5.16 - Diagrama UML representando a estrutura do XML que descreve informações a respeito do Projeto de Hardware.



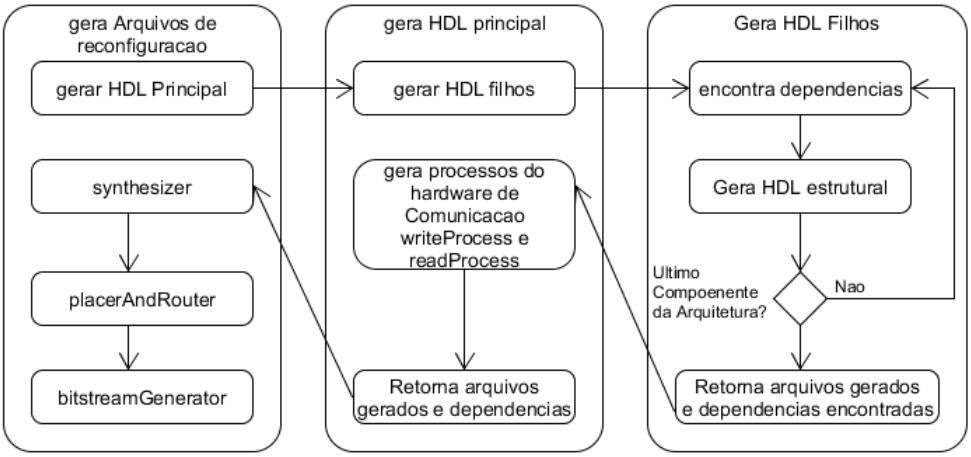
Fonte: Elaborada pelo Autor.

A Figura 5.16 ilustra a estrutura de representação do *Projeto de Hardware* dentro do *Módulo de Geração de Hardware*. É possível observar que os componentes principais, necessários para que a geração dos *bitstreams* seja feita, são uma Região Reconfigurável(*ReconfigurableRegion*), associada com uma Arquitetura(*Architecture*) e o Hardware de Comunicação(*CommunicationHardware*) correspondente.

Consultando essa estrutura o *Módulo de Geração de Hardware* gera automaticamente as entidades em HDL, que traduzem diretamente o modelo representado pela *Arquitetura de Hardware* em interconexões que podem ser interpretadas pelas Ferramentas do Fabricante.

Além da geração das interconexões em HDL, o *Módulo de Geração de Hardware* usa as informações contidas na representação do XML que representa o *Projeto de Hardware* para realizar a criação automática do módulo *Hardware de Comunicação* presente no Cliente AHM.

Figura 5.17 - Diagrama representando os principais passos da rotina de geração de *bitstreams* de reconfiguração para Arquiteturas de Hardware.



Fonte: Elaborada pelo Autor.

A Figura 5.17 contém um diagrama de estrados mostrando os principais passos do algoritmo de geração automática de hardware usado no *Módulo de Geração de Hardware*. O algoritmo consiste em processar a *Arquitetura de Hardware* fornecida a fim de realizar os seguintes procedimentos: encontrar todas os arquivos HDL dos quais os *Componentes de Hardware* presentes na arquitetura dependem; gerar entidades em HDL que venham a ser necessárias para interconexão de *Componentes de Hardware* referenciados; gerar os processos de escrita e leitura do *Hardware de Comunicação*, através do conhecimento das interfaces que precisem ser acessadas via software; e, finalmente, gerar os arquivos de *bistream* e de descrição necessários para que a Arquitetura seja implementada na Região Reconfigurável. Maiores detalhes a respeito do algoritmo são mostrados no APÊNDICE B.

5.7 MENSAGENS DE GERENCIAMENTO ENTRE O CLIENTE E REPOSITÓRIO

Para manter informações sobre o sistema embarcado atualizadas no Cliente e no Repositório AHM, algumas mensagens são trocadas. O Quadro 5.1 resume todas as mensagens de gerenciamento, bem como os momentos em que cada uma delas é necessária, uma ilustração da ordem das mensagens, no sistema em funcionamento, é mostrada na Figura 5.18.

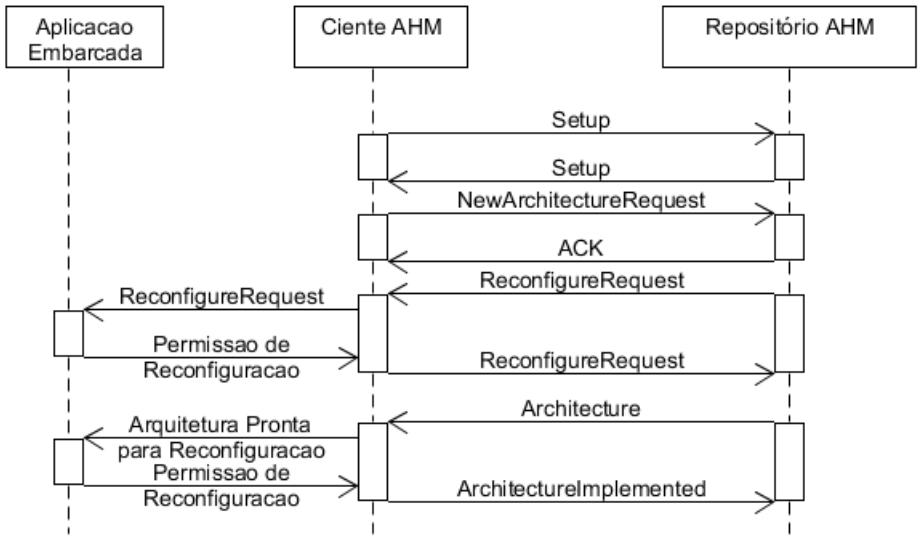
Quadro 5.1 - Mensagens de gerenciamento trocadas entre Cliente e Repositório AHM

Mensagem	Origem	Conteúdo
Setup	• *Cliente AHM*: Quando o Cliente entra em operação, essa mensagem é enviada contendo, principalmente, uma URI apontando para o arquivo de descrição do projeto relativo à esse Cliente. • *Repositório AHM*: Após receber essa mensagem do Cliente, o repositório envia a mesma mensagem, com conteúdo vazio, para terminar	• *<int, string>* ou *<vazio>* • Tamanho máximo do pacote suportado pelo Cliente AHM • Uri contendo o arquivo de descrição de arquitetura, como mostrado no projeto do sistema cliente
New Architecture Request	• *Cliente AHM*: Quando o Cliente precisa gerar o *bitstream* para uma arquitetura.	• *<txt>* • Descrição da arquitetura como mostrado nas Figura 5.5.a e Figura 5.5.b
Reconfigure Request	• *Repositório AHM:* Quando o repositório terminou de preparar uma nova arquitetura para o cliente. Nesse caso, o conteúdo da mensagem é a descrição da arquitetura.	• *<txt>* ou <vazio> • Descrição da arquitetura como mostrado nas Figura 5.5.a e Figura 5.5.b.

Mensagem	Origem	Conteúdo
Architecture	• *Repositório AHM*: Quando o Cliente informa que está preparado para receber uma nova arquitetura, o Repositório envia essa mensagem contendo os dados relativos à arquitetura.	• *<txt,bin>* • Descrição da arquitetura como mostrado nas Figura 5.5.a e Figura 5.5.b. • *bitstream* relativo à arquitetura.
Architecture Implemented	• *Cliente AHM*: Quando uma arquitetura é configurada em alguma região reconfigurável, o cliente avisa ao Repositório com essa mensagem a fim de manter os dois atualizados em termos arquiteturais.	• *<int,int>* • Uid da arquitetura implementada • Id da região reconfigurável.

Fonte: Elaborado pelo Autor.

Figura 5.18 - Diagrama de sequência contendo a ordem em que as mensagens descritas acontecem.



Fonte: Elaborada pelo Autor.

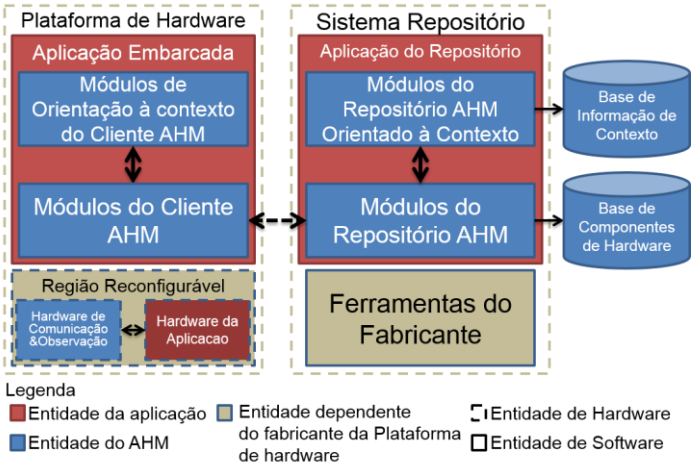
5.8 ARQUITETURA PROPOSTA ORIENTADA A CONTEXTO

Na sessão anterior foi apresentada uma arquitetura de Repositório que possibilita o gerenciamento e a geração de arquiteturas de hardware de forma automática. Nesta seção, iremos propor adições à arquitetura anterior, com o intuito de implementar um repositório sensível a contexto. Desta forma, a geração de novas arquiteturas para o Sistema Cliente levará em consideração o seu *Contexto de Operação*.

Podemos caracterizar um repositório orientado a contexto, no escopo dessa tese, como sendo um repositório que realize leituras do contexto de operação de um *Sistema Cliente* e tome decisões pré-programadas acerca da arquitetura do mesmo, em função do seu contexto.

Como pode ser visto na Figura 5.19, considerando-se a arquitetura original do repositório, três novas entidades aparecem para suportar a orientação ao contexto. As entidades *Módulos de Orientação a Contexto do Cliente AHM*, *Módulos do Repositório AHM Orientado a Contexto*, e Base de Informação de Contexto usam as APIs e mecanismos fornecidos pelos módulos do AHM, apresentados anteriormente, para implementar aplicações de hardware que podem se adaptar mediante a mudanças esperadas no *Contexto de Operação*. Além disso, o módulo *Hardware de Comunicação* passa a ter funções de monitoramento, em adição às funções de comunicação que já possuía originalmente.

Figura 5.19 - Arquitetura cliente-repositório orientada ao contexto.

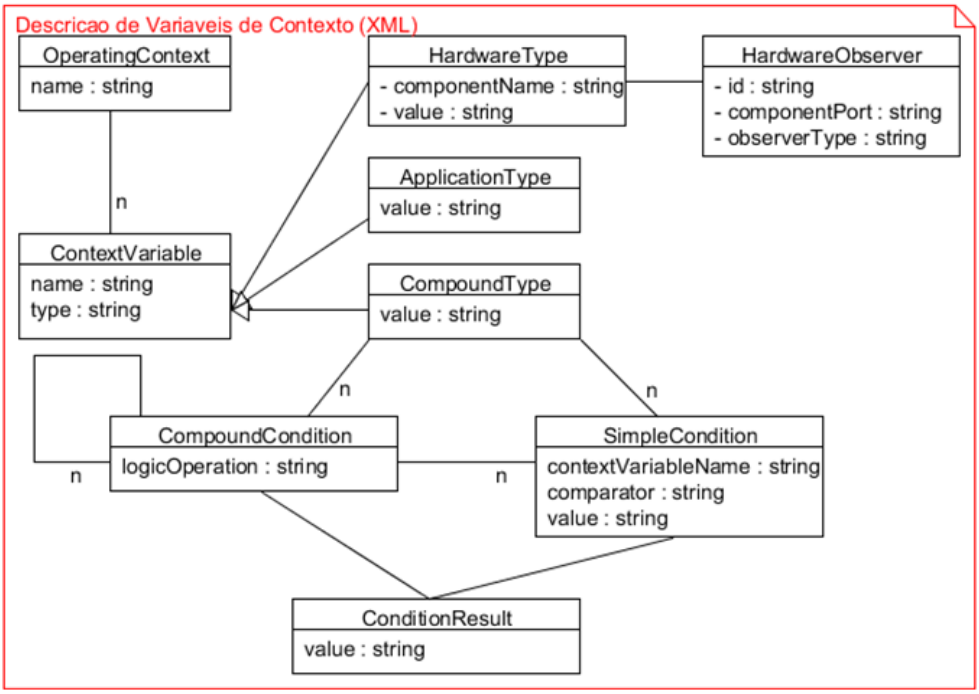


Fonte: Elaborada pelo Autor.

5.9 CONTEXTO DE OPERAÇÃO

No escopo desta tese, a representação do *Contexto de Operação* para o AHM é feita através de um documento XML que respeita a estrutura mostrada na Figura 5.20. Nesta representação, as variáveis de contexto são classificadas em três tipos: *ApplicationType*, *HardwareType* e *CompoundType*. Variáveis do tipo *ApplicationType* possuem valores que são fornecidos pela aplicação através de APIs no Repositório ou no Cliente AHM. Variáveis do tipo *HardwareType* possuem valores provenientes de interfaces de hardware que são observadas através de um componente de hardware especial chamado *HardwareObserver*. Já as variáveis do tipo *CompoundType* possuem seus valores calculados através da avaliação de operações lógicas executadas sobre os valores atuais de outras Variáveis de Contexto. Um conjunto dessas variáveis forma o contexto de operação representado pela *tag OperatingContext*.

Figura 5.20 - Diagrama UML ilustrando a estrutura do XML que representa o Contexto de Operação.

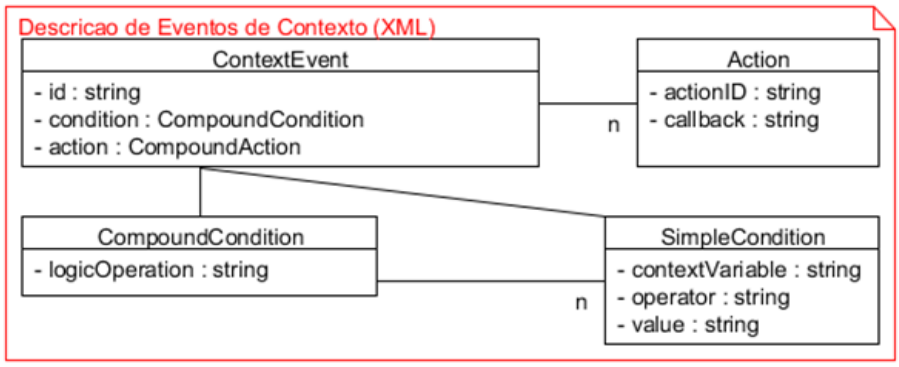


Fonte: Elaborada pelo Autor.

Uma vez definidas as variáveis do contexto, precisou-se definir regras a serem utilizadas para promover a adaptação do *Sistema Cliente*. Para tanto,

adotamos o modelo *Event-Condition-Action* (ECA) (DITTRICH, GATZIU e GEPPERT, 1995). Nesse caso, especificamos um arquivo XML para definição dos eventos, que é ilustrado na Figura 5.21, e contém as ações a serem executadas quando determinados Contextos de Operação forem alcançados.

Figura 5.21 - Diagrama UML representado a estrutura do XML que define os eventos relativos às variáveis de contexto.



Fonte: Elaborada pelo Autor.

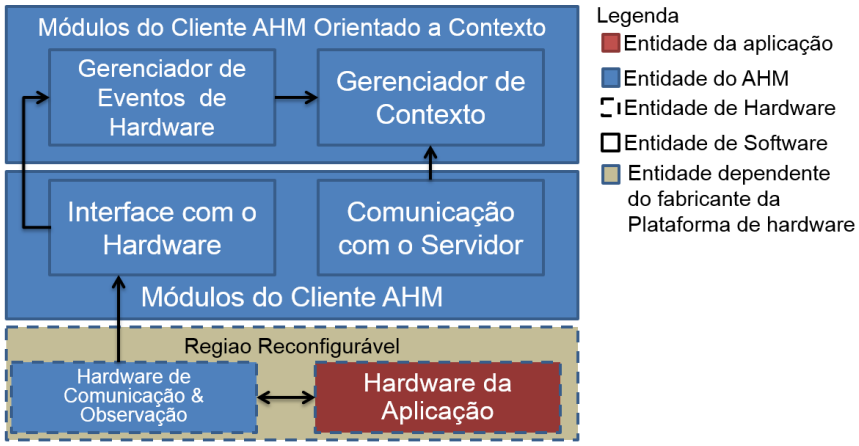
O arquivo XML de definição dos eventos permite que o desenvolvedor use relações entre as variáveis de contexto definidas para um determinado *Sistema Cliente* para disparar eventos quando o contexto de operação atinja certos estados. Quando um evento é disparado, um *callback* pode ser chamado a fim de realizar os tratamentos que sejam necessários ao evento.

Os *callbacks* são registrados na arquitetura usando as APIs fornecidas pelo *Repositório AHM*. No entanto, os eventos podem ser tratados diretamente em software, sem necessidade do registro de *callbacks*. Ambas as informações de eventos e descrições das variáveis de contexto, pertinentes ao sistema, são guardadas dentro do *Banco de Dados de Informação de Contexto*.

5.10 MÓDULOS DO CLIENTE ORIENTADO AO CONTEXTO

Para suportar a técnica de orientação a contexto, precisam ser adicionados ao *Sistema Cliente*, novos elementos que sejam capazes de monitorar os componentes de hardware e agir caso alguma alteração significativa no contexto de execução seja observada.

Figura 5.22 - módulos adicionais do Cliente AHM orientado a contexto.



Fonte: Elaborada pelo Autor.

Os módulos representados na

Figura 5.22 cobrem o requisito de geração e processamento dos eventos de reconfiguração:

- *Módulo Gerador de Eventos de Hardware*: É responsável por interagir com o *Módulo de Interface com o Hardware* para detectar variações no contexto de operação usando os componentes observadores de hardware.
- *Módulo Gerenciador de Contexto*: Responsável por guardar a representação contextual da *Aplicação Embarcada* cliente. Esse módulo usa as interfaces de comunicação para informar ao *Repositório AHM* dados a respeito do estado atual das variáveis de contexto do sistema.

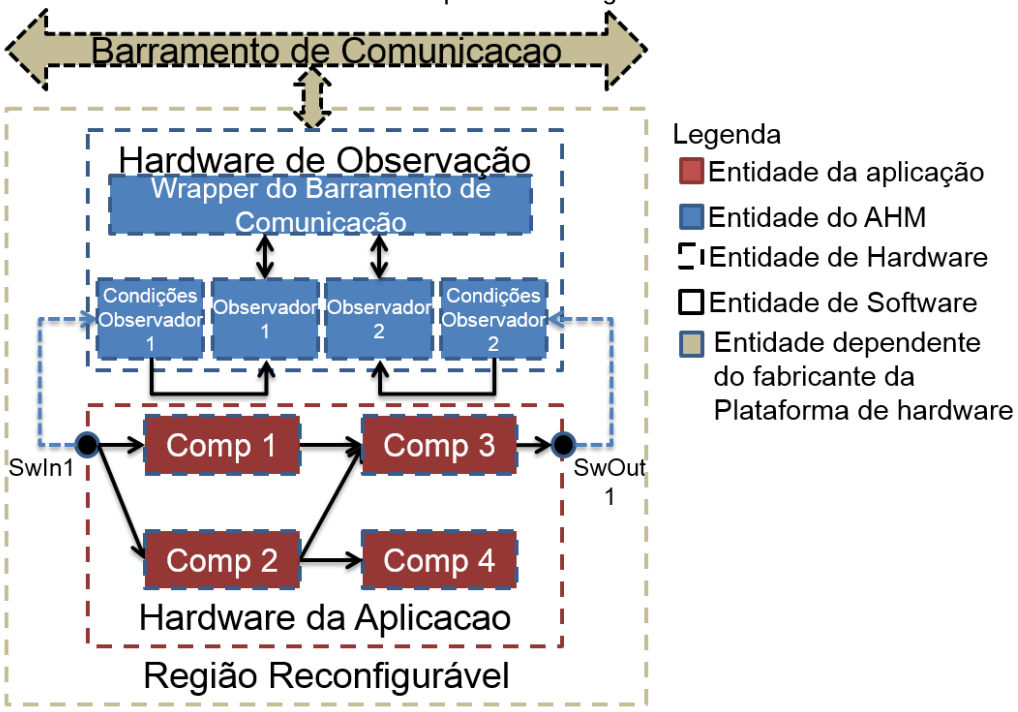
Um fato importante é que a representação do *Contexto de Operação de um Sistema Cliente* pode variar com o tempo, uma vez que novas arquiteturas de hardware, com outras *Variáveis de Contexto*, podem ser configuradas no sistema. Por isso, os módulos do cliente são focados em monitoramento enquanto os elementos de tomada de decisão se localizam no Repositório.

5.11 HARDWARE DE OBSERVAÇÃO

Usando as soluções de geração de componentes de hardware fornecidas pelo *Repositório AHM* é possível criar, em paralelo com a criação do *Hardware de*

Comunicação, componentes de hardware que realizam observação de entradas ou saídas de subcomponentes presentes na arquitetura de hardware gerenciada.

Figura 5.23 - Esquema desenvolvido para observação dos componentes de hardware executando na arquitetura configurada.



Fonte: Elaborada pelo Autor.

A Figura 5.23 ilustra como é feita a inserção do *Hardware de Observação e Comunicação* na arquitetura de hardware configurada na *Aplicação Embarcada*. Este componente por sua vez é composto, além do Hardware de Comunicação presente originalmente no AHM, por dois subcomponentes: *Componente de Condições*, e *Componente Observador*.

O *Componente Observador* é uma especialização das *Interfaces Genéricas*, apresentadas na Seção 5.4. *Observadores* são interfaces de apenas leitura e as informações providas pelos mesmos não são apenas um espelho da informação presente na entrada ou saída de outros componentes que estão sendo monitorados, mas sim a computação da informação de contexto relativa àquela interface.

Quadro 5.2 - Operadores e tipos de Observador suportados pelo Hardware de Observação.

Operador	Tipo do Observador	Descrição
gt(>), gte(>=), lt(<), lte(<=), eq(==)	valor	Valor do observador = 1, caso a interface monitorada satisfaça o operador especificado usando o atributo valor. Ex: interface1 gt valor; interface2 eq valor.
gt(>), gte(>=), lt(<), lte(<=), eq(==)	interface	Valor do observador é 1, caso a interface monitorada satisfaça o operador especificado usando o valor de outra interface.
and, or, nand, nor	observer	Valor do observador é calculado, usando a expressão: (observerIndex1 op observerIndex2). Onde observerIndex são identificadores para outros observadores presentes nesta

Fonte: Elaborada pelo Autor.

No *Componente de Condições*, é possível definir um conjunto de operadores lógicos e/ou algébricos, listados no Quadro 5.2, que irão processar os dados observados nas interfaces monitoradas do *Hardware da Aplicação*. O resultado do processamento é representado através de valores lógicos e armazenado nos *Observadores*, portanto os módulos de monitoramento de contexto do AHM no nível de software precisam apenas checar, no *Observador*, se a condição foi satisfeita. Dessa forma, a maior parte do overhead envolvido na observação de *Variáveis de Contexto* do tipo *HardwareType* que, comumente, possuem alta frequência de atualização fica implementado nos Observadores.

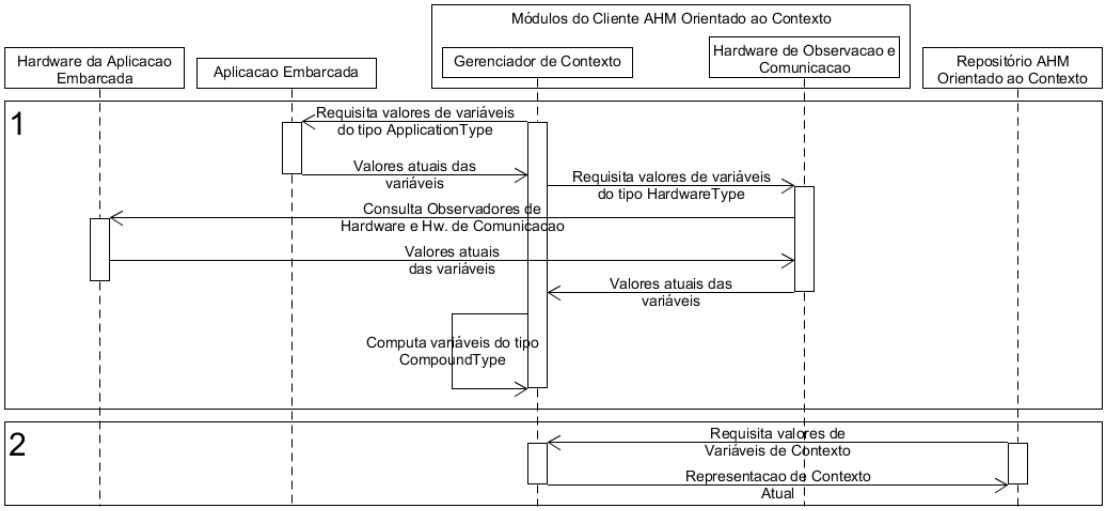
O módulo de software *Gerenciador de Eventos de Hardware* executa em paralelo aos módulos do *Cliente AHM* e tem o objetivo de checar os estados de todos os elementos do componente *Hardware de Observação*, gerando eventos de software quando acontece transição de valores em algum deles.

5.12 FUNCIONALIDADES IMPLEMENTADAS NO CLIENTE AHM ORIENTADO AO CONTEXTO

Os módulos mostrados, junto com os *Hardware de Observação* implementam as principais funcionalidades de monitoramento e interação com o *Contexto de Operação* da *Aplicação Embarcada*.

As principais funcionalidades implementadas nestes módulos são: Monitoramento do Contexto de operação de acordo com o tipo de variável de contexto monitorada; Atualização do modelo de contexto local; atender requisições de monitoramento do advindas do Repositório; e gerar *Eventos de Contexto* que venham a ocorrer e enviá-los ao Repositório.

Figura 5.24 - 1) Processo de monitoramento automático de Contexto no Cliente AHM Orientado à Contexto; 2) Processo de atualização da representação de Contexto do Repositório AHM.

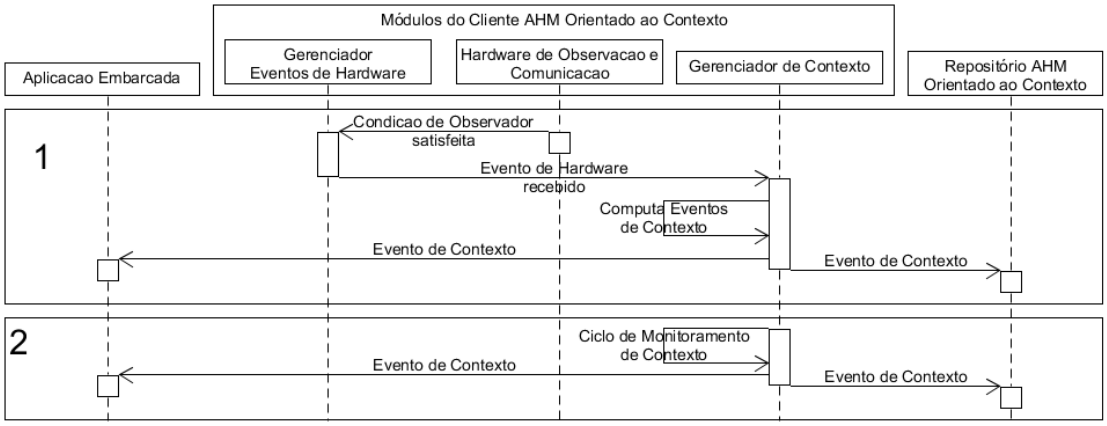


Fonte: Elaborada pelo Autor.

A Figura 5.24 apresenta os diagramas de sequência dos processos de monitoramento do *Contexto de Operação* da aplicação seguindo o modelo adotado pelo *AHM Orientado à Contexto*. O monitoramento se dá em três passos, ilustrados na Figura 5.24.1, inicialmente o *Gerenciador de Contexto* atualiza as variáveis de contexto do tipo *ApplicationType*, chamando *callbacks* implementados pela própria aplicação embarcada. Em seguida, as variáveis do tipo *HardwareType* são atualizadas através da consulta dos *Hardware de Observação e Comunicação*. Os valores recebidos das variáveis são, então, usados para atualizar as variáveis de contexto do tipo *CompoundType*, completando assim o ciclo de monitoramento

no *Cliente AHM Orientado ao Contexto*. Outro fluxo de monitoramento representado pela Figura 5.24.2, consiste na atualização da representação contextual presente no *Repositório AHM*, neste caso, o módulo *Gerenciador de Contexto* recebe a requisição do servidor, através do *Módulo de Comunicação*, e envia de volta um conjunto de valores representando o estado atual das variáveis de contexto monitoradas.

Figura 5.25 - 1) Processo de monitoramento geração de Eventos de Contexto advindos dos Módulos de Hardware; 2) Processo de geração de Eventos de Contexto advindos dos Módulos de Software.



Fonte: Elaborada pelo Autor.

A Figura 5.25 ilustra os dois processos de geração de *Eventos de Contexto* que podem ocorrer no *Cliente AHM Orientado à Contexto*. Eventos de contexto podem ser gerados de duas formas, partindo de Eventos gerados a partir dos Hardware de Observação ou partindo do processamento das variáveis de contexto advindas dos módulos de software.

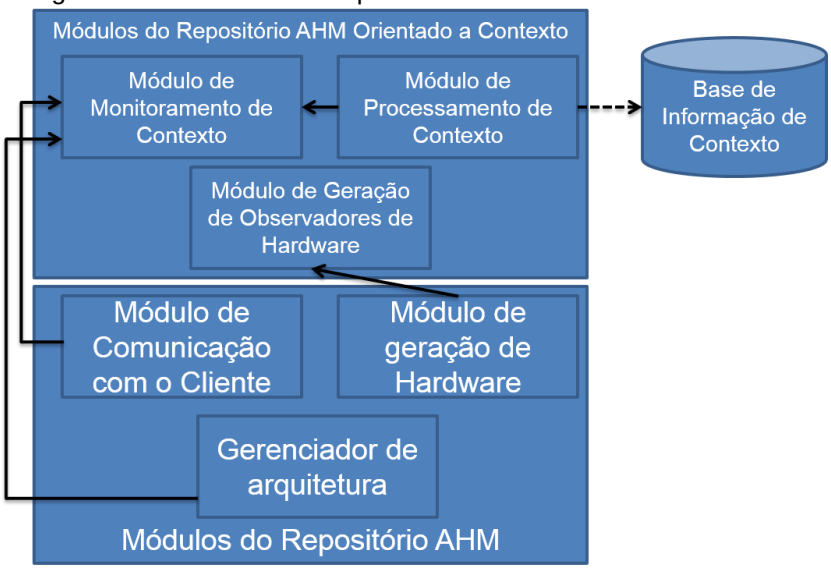
Na Figura 5.25.1 os Eventos de Contexto, relacionados diretamente às Variáveis de Contexto do tipo *HardwareType*, são originados no módulo *Hardware Observação e Comunicação*. Eventos deste tipo são percebidos pelos Observadores de Hardware quando o operador presente no componente *Condição do Observador* é satisfeito. Após isso o módulo *Gerenciador de Eventos de Hardware* é evocado para que o evento possa ser repassado para o *Gerenciador de Contexto* que processa o evento de hardware recebido podendo ou não gerar um *Evento de Contexto* correspondente.

Já fluxo de informações representado na Figura 5.25.2 diz respeito aos *Eventos de Contexto* que são gerados a partir de outras interações do módulo Gerenciador de Contexto como o processamento de variáveis de contexto do tipo *ApplicationType* ou *CompoundType*.

5.13 MÓDULOS DO REPOSITÓRIO AHM ORIENTADO AO CONTEXTO

Para implementação dos componentes de sensibilidade a contexto no Repositório, algumas novas entidades foram adicionadas à arquitetura, com o intuito de possibilitar a geração dos observadores de hardware, realizar o monitoramento das variáveis de contexto e disparar os eventos de contexto configurados.

Figura 5.26 - Módulos do repositório AHM orientado ao contexto.



Fonte: Elaborada pelo Autor.

A Figura 5.26 mostra os componentes adicionados ao sistema Repositório. Um detalhamento de cada módulo é dado abaixo:

- *Módulo de Monitoramento de contexto*: Esse módulo é responsável por realizar a atualização das variáveis de contexto configuradas usando os módulos de comunicação e a lista de variáveis descritas para o sistema em operação. Esse módulo também é responsável por disparar

possíveis eventos de contexto configurados, fazendo chamadas aos devidos callbacks, quando especificado.

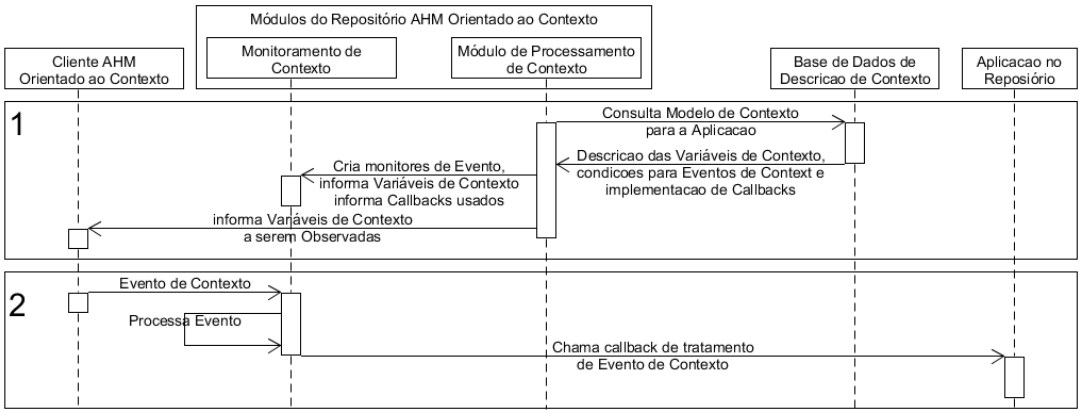
- *Módulo de Processamento de Contexto*: Este módulo representa o principal componente do Repositório orientado a contexto. Ele se conecta à base de dados que contém a descrição da representação de contexto usada para a Aplicação Embarcada bem como as descrições para os eventos de contexto. Usando a informação contida na base de dados, esse módulo constrói os gatilhos de eventos e estruturas de monitoramento que são usadas para a observação do Contexto de Operação da Aplicação Embarcada.
- *Módulo de Geração de Observadores de Hardware*: Módulo responsável por adicionar os observadores de hardware caso alguma interface de hardware precise ser monitorada.

5.14 FUNCIONALIDADES PROVIDAS PELO REPOSITÓRIO AHM ORIENTADO AO CONTEXTO.

As funcionalidades providas pelo *Repositório AHM* Orientado ao Contexto complementam as que já foram implementadas no *Cliente AHM* visando possibilitar o gerenciamento automático através dos parâmetros fornecidos pelo Desenvolvedor da Aplicação: descrição de *Variáveis de Contexto*, descrição de condições para geração dos *Eventos de Contexto* e descrição dos *callbacks* a serem chamados no caso da ocorrência dos eventos descritos.

As principais funcionalidades implementadas nestes módulos são: Fornecer ao *Cliente AHM* o modelo de descrição de Contexto de Operação e Eventos de Contexto adotado pelo Repositório; Monitoramento do modelo de contexto no *Cliente AHM*; Processamento dos eventos de contexto gerados no escopo do *Cliente AHM*; Gerar os componentes de *Hardware de Observação* em conjunto com o *Hardware de Comunicação*, de acordo com as Variáveis de Contexto que precisem ser observadas.

Figura 5.27 - 1) Diagrama de sequência para criação das estruturas de monitoramento de Contexto de Operação; 2) Diagrama de sequência para monitoramento e tratamento de Eventos de Contexto.

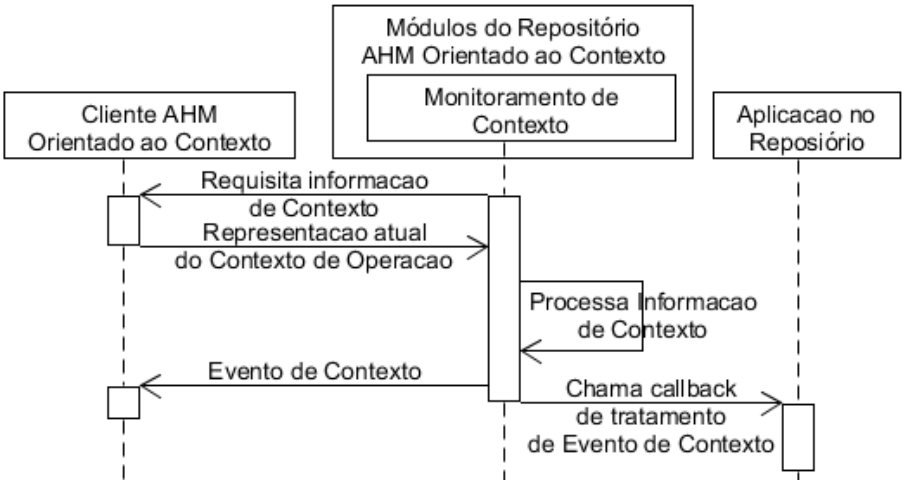


Fonte: Elaborada pelo Autor.

A Figura 5.27 ilustra os diagramas de sequência relacionados com a criação das estruturas de monitoramento e tratamento de *Eventos de Contexto*. Na primeira parte, correspondente à Figura 5.27.1, temos o fluxo de dados que ocorre durante a inicialização dos componentes de observação e descrição de *Contexto de Operação*. As informações em XML que descrevem as variáveis de contexto presentes na Aplicação bem como as condições necessárias para geração de eventos de contexto ficam armazenadas no *Banco de Dados de Descrição de Contexto*, o *Módulo de Processamento de Contexto* é responsável por gerenciar essas informações e realizar o setup dos *Módulos de Monitoramento de Contexto* tanto do Repositório quanto do Cliente AHM. Este fluxo de dados também pode ocorrer caso haja uma modificação na representação de Contexto da Aplicação.

A segunda parte da Figura 5.27.2, pode ocorrer quando os componentes de monitoramento estão devidamente configurados. Após isso, *Eventos de Contexto* recebidos pelo sistema são tratados pelo *Módulo de Monitoramento de Contexto*, que é responsável por chamar os *callbacks* de tratamento de evento devidamente registrados pela *Aplicação no Repositório*.

Figura 5.28 - Diagrama de sequência representando a atualização dos valores das Variáveis de Contexto Repositório AHM.

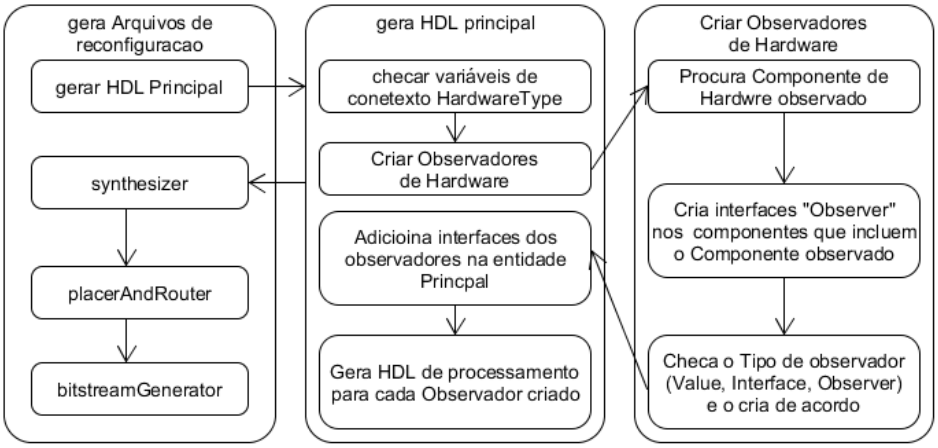


Fonte: Elaborada pelo Autor.

A Figura 5.28 ilustra como é feita a atualização das *Variáveis de Contexto* pelo *Repositório AHM Orientado ao Contexto*. Como pode ser visto na figura, o Repositório requisita os valores das *Variáveis de Contexto*. Após isso o Cliente AHM envia uma mensagem contendo os valores atuais das variáveis observadas, com os valores atualizados, o *Módulo de Monitoramento de Contexto* processa as variáveis e, no caso em que as condições para algum evento sejam satisfeitas, dispara algum *Evento de Contexto* correspondente.

A última funcionalidade do *Repositório AHM Orientado ao Contexto*, diz respeito à geração dos componentes *Hardware de Observação*. A geração desses componentes de hardware é feita usando o mesmo fluxo de dados mostrado, na Figura 5.15, para geração automática de hardware. No entanto, nesse caso, é preciso que seja levado em consideração as variáveis de contexto que precisam ser observadas, bem como os eventos relacionados a essas variáveis.

Figura 5.29 - Passos do algoritmo de geração e Observadores de Hardware.



Fonte: Elaborada pelo Autor.

A Figura 5.29 mostra os passos para geração automática do componente *Hardware de Observação*. O algoritmo de geração, inicialmente usado pelo AHM foi estendido para incluir a geração dos *Hardwares de Observação*, bem como os *Hardwares Condição do Observador*.

O algoritmo começa procurando, na representação atual, variáveis de contexto do tipo *HardwareType*, para cada variável desse tipo, um componente do tipo *Observador* será criado. Para que seja possível observar uma interface de um subcomponente da *Arquitetura de Hardware*, é externar essa interface no componente principal da arquitetura. Isso é feito através da criação de interfaces em todos os subcomponentes que incluem, direta ou indiretamente, o componente observado.

Após isso, a representação das condições para os *Eventos de Contexto* é consultada para que seja possível determinar qual tipo de componente *Condição do Observador*(do tipo Valor, Observador ou Interface) será criado.

Os HDL para o *Hardwares de Observação* criados são, então, adicionados à lista de dependências da arquitetura de hardware e uma interface de acesso via software para o *Observador* é criada. Após esse último passo, a geração de *Arquiteturas de Hardware* prossegue como mostrado, anteriormente, na Figura 5.17. Maiores detalhes a respeito do algoritmo de geração dos *Hardwares de Observação* são dados no Apêndice C.

5.15 MENSAGENS DE GERENCIAMENTO ENTRE O CLIENTE AHM ORIENTADO À CONTEXTO E O REPOSITÓRIO AHM ORIENTADO À CONTEXTO

Para realização da atualização das variáveis de contexto e acompanhamento das transições ocorridas no cliente, o Repositório AHM usa um conjunto de mensagens de atualização, semelhante às mensagens de gerenciamento mostradas anteriormente. A lista de mensagens é exibida na Quadro 5.3.

Quadro 5.3 - Lista de mensagens de atualização de variáveis de Contexto.

Mensagem	Origem	Conteúdo
ObservationDataRequest	• *Repositório AHM*: O repositório envia essa mensagem ao servidor informando a lista de variáveis de contexto que pretende atualizar nesse ciclo.	• Lista de elementos *<string *>* • Nome da variável de contexto
ObservationDataResponse	• *Cliente AHM*: Ao receber uma mensagem do tipo ObservationDataRequest o cliente responde com os valores das variáveis de contexto que possui em sua representação.	• Listade elementos *<string, string*>* • Nome da variável e valor atual da variável de contexto

Mensagem	Origem	Conteúdo
ContextEvent	• *Repositório AHM*: O repositório envia essa mensagem para notificar ao cliente que um evento de contexto aconteceu. • *Cliente AHM*: O cliente envia essa mensagem para notificar ao repositório que um evento	• *<string>* • Id do evento que aconteceu

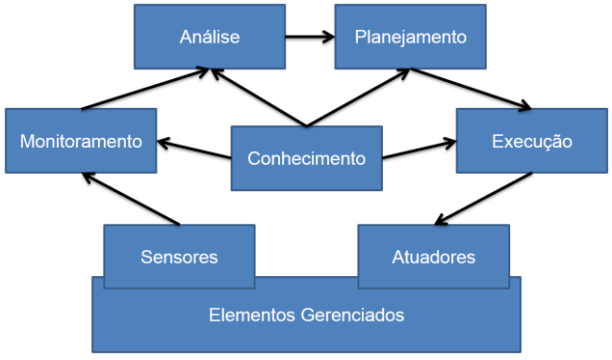
Fonte: Elaborada pelo Autor.

5.16 ARQUITETURA AHM ORIENTADA AO CONTEXTO E O MODELO DE SISTEMAS AUTONÔMICOS

Apresentado na seção de trabalhos relacionados, o modelo de sistemas autonômicos é baseado no modelo MAPE-K. Portanto, para que um sistema seja considerado Autonômico, é necessário que o mesmo se comporte com base nesse modelo.

Embora as soluções apresentadas na seção de trabalhos relacionados implementem o modelo MAPE-K, nenhuma delas utiliza um repositório de componentes de hardware. Haja vista o modelo arquitetural do Repositório AHM apresentado até agora, é possível observar a presença de vários elementos do modelo MAPE-K.

Figura 5.30 - modelo MAPE-K usado para sistemas autonômicos.



Fonte: Elaborada pelo Autor.

A Figura 5.30 ilustra o modelo MAPE-K usado para sistemas autonômicos. Um possível conjunto de relações entre os módulos do modelo MAPE-K e as soluções adotadas no Repositório AHM orientado a contexto é mostrado no Quadro 5.4.

Com exceção da Análise e parte do Planejamento, que são dependentes da aplicação, os outros componentes são completamente mapeados. Usando um repositório, podemos ainda contar com as vantagens da reflexão da arquitetura de hardware, devido ao uso do modelo de componentes, bem como maior flexibilidade de otimização, uma vez que podemos gerar arquiteturas de hardware durante a execução do sistema.

Neste capítulo mostramos a concepção e implementação de um repositório ativo orientado a contexto para componentes de hardware, o AHM Orientado a contexto. Ao fim tentamos relacionar os módulos do AHM com o modelo MAPE- K usado para implementação de sistemas autonômicos. Para comprovar a Tese de que um repositório ativo pode ser usado na implementação de sistemas autonômicos, aplicaremos os conceitos e soluções mostradas nesta seção para implementar sistemas autonômicos no Capítulo 6.

Quadro 5.4 - mapeamento das estruturas do MAPE-K e os componentes do AHM apresentados

MAPE-K	AHM Orientado ao contexto
Sensores	Módulos monitoramento de contexto e arquitetura no *Cliente AHM* e *Repositório AHM.*
Monitoramento	Módulos de gerenciamento de *Eventos de Contexto* e *Variáveis de Contexto* no repositório e cliente AHM.
Análise	Dependente de aplicação, porém pode ser inserido usando APIs providas pelo repositório e cliente AHM.
Planejamento	Módulos de criação de arquitetura de hardware, porém esse elemento do modelo também depende da aplicação.
Execução	Módulos reconfiguração de hardware no Repositório AHM.
Atuadores	Módulos de reconfiguração no cliente AHM.
Conhecimento	Modelo de contexto, eventos de contexto e *callbacks* de reconfiguração de arquitetura.

Fonte: Elaborada pelo Autor.

CAPÍTULO 6
EXPERIMENTOS E RESULTADOS

Para análise da viabilidade da arquitetura proposta, estudos foram realizados com o intuito de comprovar as hipóteses de pesquisa desta tese e cumprir os objetivos apresentados no CAPÍTULO 1 de forma sistemática. Neste capítulo, apresentamos os detalhes de implementação de algumas aplicações de teste, os experimentos realizados e seus respectivos resultados, propostos com o intuito de testar as características principais do AHM. Inicialmente, uma aplicação denominada *Hardware Reconfigurable Filter* foi desenvolvida com vistas a testar a capacidade configuração e gerenciamento do AHM. Uma segunda aplicação desenvolvida foi a *Autonomic Image Segmentation System*, um sistema autonômico de processamento digital de imagens cujas características autonômicas foram implementadas usando a arquitetura AHM; por último, a fim de demonstrar que a arquitetura AHM pode ser aplicada em conjunto a outros sistemas encontrados no mercado, apresentaremos a aplicação *Autonomic Auto Pilot*, um sistema autonômico que implementa os controladores de um veículo aéreo não tripulado usando a arquitetura AHM.

6.1 EXPERIMENTO 1: HARDWARE RECONFIGURABLE FILTER

Para verificar a viabilidade de manipulação e geração, sob demanda, de arquiteturas de hardware e sua instanciação em plataformas reconfiguráveis usando o esquema cliente-repositório, uma aplicação denominada *Hardware Reconfigurable Filter* foi desenvolvida. Esta aplicação de testes é proposta com o objetivo de validar as hipóteses secundárias HS1, HS2, HS3 e HS4, listadas no Capítulo 1. Para tanto, o seguinte experimento foi realizado:

Experimento: Desenvolver uma aplicação de hardware reconfigurável utilizando a solução Cliente-Repositório proposta.

Objetivo Principal: Testar os módulos básicos do AHM, validando as hipóteses secundárias listadas no Capítulo 1:

HS1 - *É possível adaptar um modelo de componentes de software para que possa ser aplicado a componentes de hardware.*

HS2 - *O sistema de hardware pode ser reconfigurado, dentro do sistema embarcado, através de interfaces de software.*

HS3 - *É possível gerar novas arquiteturas de hardware para o sistema gerenciado, usando componentes disponíveis, um engenho de software e as ferramentas de síntese de hardware providas pelo fabricante.*

HS4 - *É possível desenvolver uma arquitetura de hardware embarcado em dispositivo reconfigurável na qual seja possível observar o estado de cada componente da arquitetura, independentemente da quantidade de componentes e interfaces presentes na mesma.*

Para nortear a realização do experimento, os objetivos secundários abaixo foram estabelecidos:

OS1- Projetar uma aplicação de filtro digital de sinais, que possibilite a utilização dos componentes da arquitetura AHM.

OS2- Projetar e representar os componentes da aplicação, usando o modelo de componentes escolhido pelo AHM.

OS3- Testar a geração automática de arquiteturas de hardware para a aplicação desenvolvida em OS1, usando as interfaces providas pelo AHM.

OS4- Testar a reconfiguração e gerenciamento do hardware embarcado usando as APIs providas.

OS5- Testar o sistema de acesso ao hardware gerado usando a API de comunicação provida pelo AHM.

Para consecução dos objetivos secundários da aplicação Hardware *Reconfigurable Filter*, o conjunto de requisitos funcionais abaixo foi estabelecido:

R1- O sinal filtrado deverá ser lido e provido ao hardware por meio da aplicação embarcada. Suas componentes de frequência principais devem ser providas pelo operador do sistema.

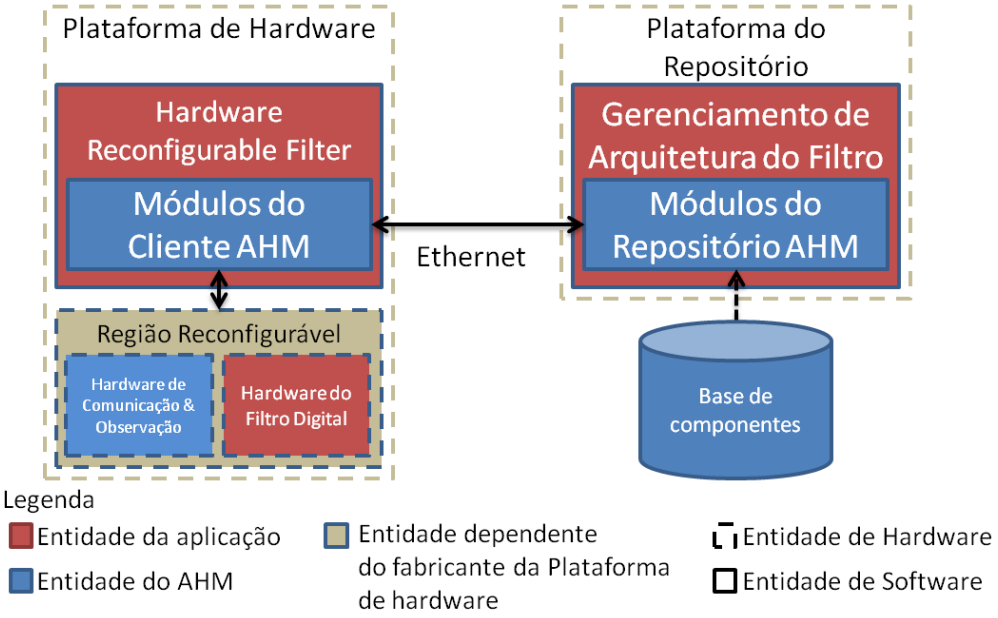
R2- Componentes de frequência não centradas nas frequências principais do sinal devem ser consideradas ruído e filtradas, se possível.

R3- Os valores desejados ordem máxima e mínima dos filtros que compõem o sistema podem mudar com o tempo.

R4- A ordem do filtro deve ser aumentada em 1 se o sinal filtrado contiver componentes de ruído com uma relação de amplitude maior do que um *threshold* fornecido pelo operador do sistema.

Considerando-se os requisitos estabelecidos, a aplicação foi dividida em duas partes: uma Aplicação Embarcada que implementa as funcionalidades básicas do filtro, executando na Plataforma de Hardware, e outra aplicação que gera novas arquiteturas de filtro quando necessário, implementada no Repositório AHM, resultando na arquitetura orientada a Host mostrada na Figura 6.1.

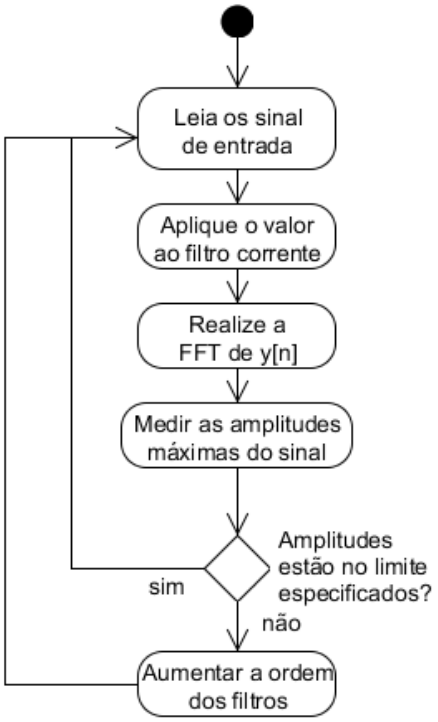
Figura 6.1 - Arquitetura da aplicação Hardware Reconfigurable Filter.



Fonte: Elaborada pelo Autor.

Com a divisão realizada, o sistema embarcado fica com dois papeis bem definidos: alimentar as entradas do filtro com os valores do sinal a ser filtrado; realizar as leituras de valores do SNR e informar ao repositório. A *Aplicação Embarcada* implementa, portanto, o requisito R3, enquanto os demais requisitos executarão no repositório, implementando o diagrama mostrado na Figura 6.2.

Figura 6.2 - Aplicação implementada no repositório.



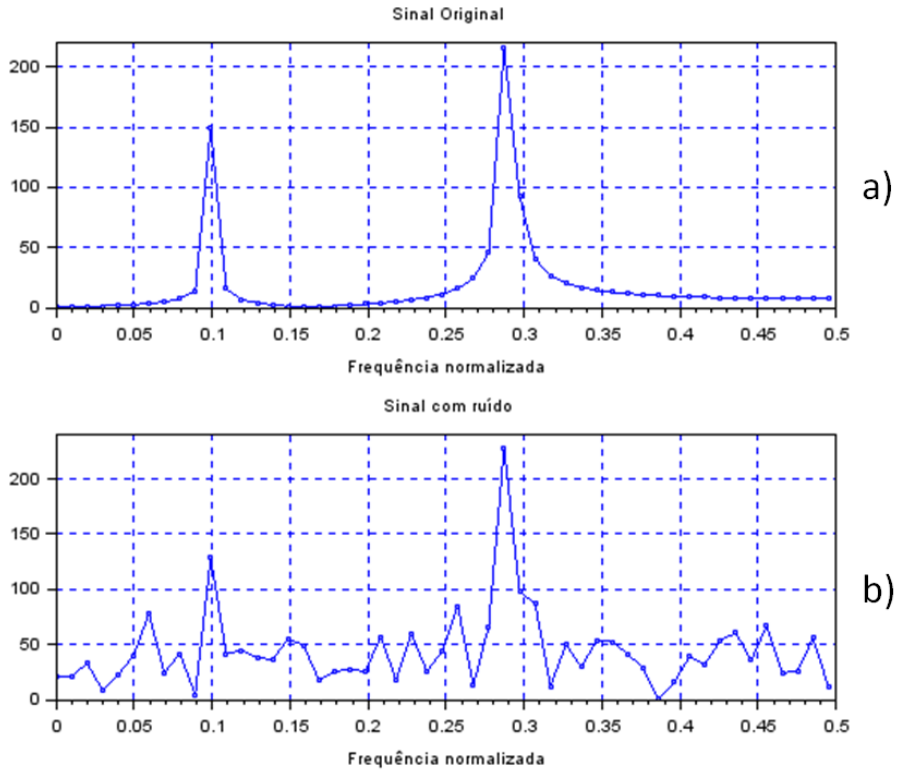
Fonte: Elaborada pelo Autor.

6.1.1 Execução do Sistema

A implementação foi realizada em uma plataforma Zedboard (AVNET CORPORATION, 2012), enquanto o Repositório e a Aplicação do Repositório foram implementados em um computador Desktop Intel Core I3, 3.0 GHz, 6.00 GB de memória RAM, executando Linux.

Enquanto a Aplicação Embarcada executa na plataforma de hardware, sinais são gerados e fornecidos via ethernet para serem filtrados pela aplicação. A Figura 6.3.a mostra o sinal a ser filtrado, sem componentes de ruído, enquanto a Figura 6.3.b mostra o mesmo sinal após adicionado de um ruído branco centrado na frequência 0.1.

Figura 6.3 - a) Representação em frequência do Sinal a ser filtrado, sem ruídos externos. b) Representação em frequência do Sinal após adição de ruídos.

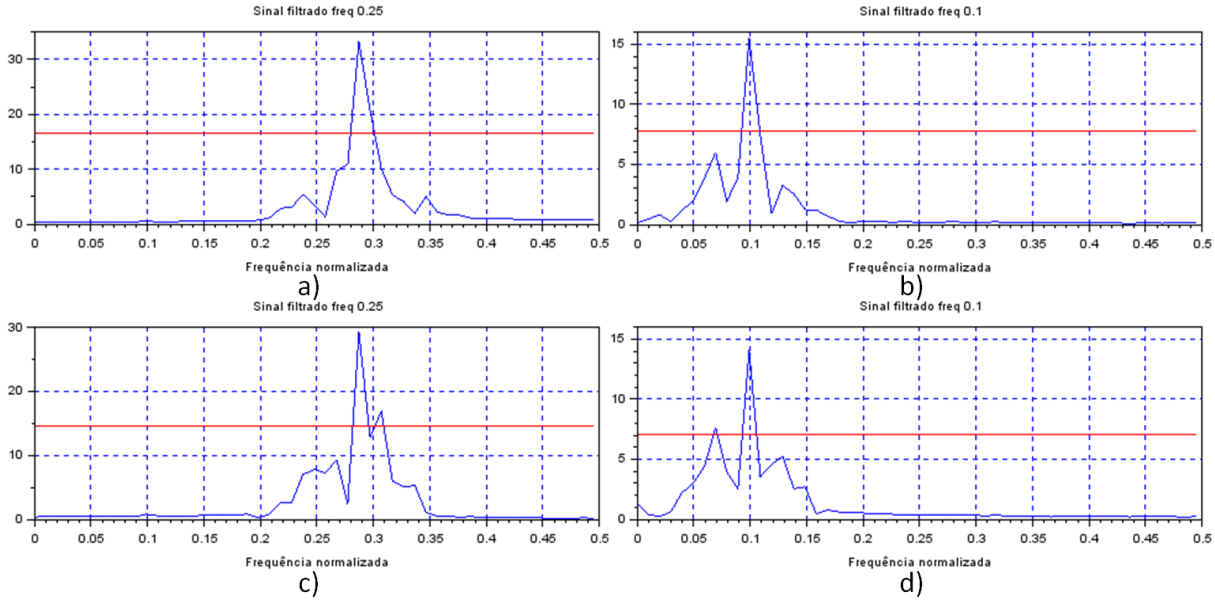


Fonte: Retirada do programa Scilab.

A aplicação executa de forma contínua recebendo os valores do Sinal x[n] e executando os passos enumerados na Figura 6.2. Os coeficientes do filtro para um dado sinal são calculados na aplicação de Gerenciamento de Arquitetura do Filtro, que executa no repositório, usando a função *wfir* (SCILAB) provida pelas APIs do Scilab. A função permite a geração dos coeficientes dos filtros cujos valores são usados para gerar novas arquiteturas de Filtro quando requisitada pela aplicação cliente.

A aplicação inicia com dois filtros de Hamming passa baixa, de ordem 33, centrados em duas faixas de frequência normalizadas, 0.1 e 0.27, representadas na Figura 6.3. O threshold fornecido inicialmente é de 0.5, o que significa que componentes de ruído com amplitude igual ou maior que metade da amplitude do sinal principal causará aumento na ordem do filtro.

Figura 6.4 - Representação em frequência dos Sinais Filtrados, linhas vermelhas representam o *threshold* escolhido para esse caso. a) e c) Sinais filtrados sem componentes de ruído aparentes. b) e d) Sinais apresentando componentes de frequência que precisam ser tratados através do aumento da ordem do filtro.



Fonte: Retirada do programa Scilab.

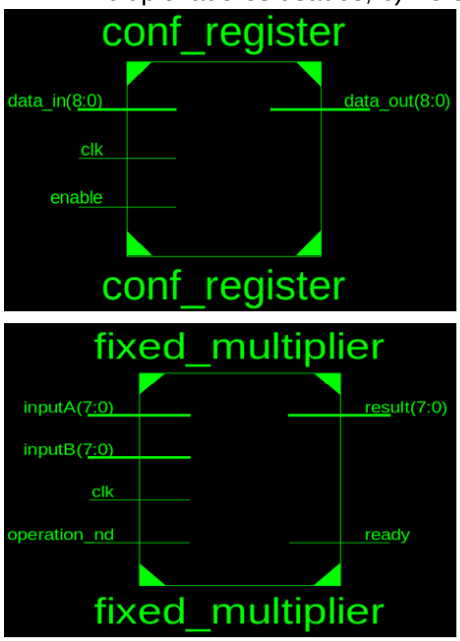
A Figura 6.4.a e b representam amostras do sinal x[n] gerado, enviado e filtrado pelo filtro digital implementado. Um ruído branco gaussiano centrado em frequências aleatórias, porém próxima das frequências de corte dos filtros é gerado e adicionado ao sinal x[n]. Essa adição pode resultar nos casos demonstrados nas Figura 6.4.c e d, onde parte do ruído passa pelo filtro com uma amplitude maior ou igual ao *threshold* configurado, nesse caso uma reconfiguração do filtro é executada, aumentando a ordem do filtro para remover a componente de ruído.

6.1.2 Análise dos resultados

Após a implementação do sistema, dados foram coletados para verificar se os objetivos foram alcançados como esperado. Em virtude do objetivo secundário OS1 ter a ver com a concepção e implementação da aplicação, iremos desconsiderá-lo nessa sub seção, pois a modelagem do sistema apresentada já apresenta êxito neste quesito independentemente dos resultados obtidos para os outros objetivos secundários.

Seguindo na lista de objetivos secundários, o objetivo OS2 é o de “Projetar e representar os componentes da aplicação, usando o modelo de componentes escolhido pelo AHM”. Essa representação, como visto na arquitetura do AHM, é composta por um arquivo de descrição do componente que reflete diretamente seu modelo em SystemC, e que é manipulado pelo Repositório.

Figura 6.5 - a) Representação em forma de Componente de Hardware dos Registradores e Multiplexadores usados; b) Versão em XML da representação dos componentes.




```
<componentBase>
  <component name="conf_register" file="conf_register.vhd">
  <param name="dataSize" type="integer" defaultValue="8">
  <input name="clk" type="bit"/>
  <input name="enable" type="bit"/>
  <input name="data_in" type="vector"
         startIndex="dataSize - 1" op="downto" endIndex="0"/>
  <output name="data_out" type="vector"
         startIndex="dataSize - 1" op="downto" endIndex="0"/>
 </component>

 <component name="fixed_mult_ip" ipCoreManufacturer="xilinx"
         ipCore="xilinx.com:ip:floating_point:5.0">
  <ipcoreParams>
   <param name="c_a_exponent_width" type="integer" defaultValue="4"/>
   <param name="c_a_fraction_width" type="integer" defaultValue="4"/>
   <param name="c_result_precision_type" type="string" defaultValue="custom"/>
   <param name="operation_type" type="string" defaultValue="multiply"/>
  </ipcoreParams>
  <input name="a" type="vector"
         startIndex="c_a_exponent_width + c_a_exponent_width - 1"
         op="downto" endIndex="0"/>
  <input name="b" type="vector"
         startIndex="c_a_exponent_width + c_a_exponent_width - 1"
         op="downto" endIndex="0"/>
  <input name="operation_nd" type="bit">
  <input name="clk" type="bit">
  <input name="scl" type="bit">
  <output name="result" type="vector"
         startIndex="c_a_exponent_width + c_a_exponent_width - 1"
         op="downto" endIndex="0"/>
  <output name="rdy" type="bit">
 </component>
</componentBase>
```

a) b)

Fonte: Elaborada pelo Autor.

A Figura 6.5 mostra a representação em XML para dois componentes da aplicação de filtro digital implementada, considerando-se o modelo de componentes adotado. Pela representação gráfica do componente, retirada da ferramenta de síntese de hardware da Xinlinx, não é possível identificar os parâmetros de configuração dos componentes que podem ser vistos na representação em XML. Usando os módulos do AHM a Aplicação Embarcada ou do Repositório podem configurar esses parâmetros, além de mudar a arquitetura de hardware.

A representação lógica dos Componentes de Hardware é gerada a partir da descrição XML fornecida. Essa representação é descrição da representação estendida usando o modelo de componentes do SystemC, por isso ela pode ser aplicada a qualquer tipo de componente de hardware que possa ser descrito usando

o modelo do SystemC. Dessa forma, concluímos que o objetivo dois do experimento foi cumprido.

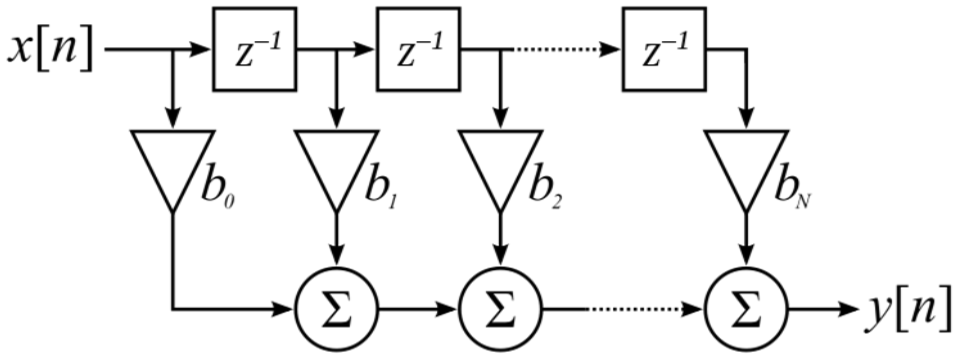
Os testes, a respeito da geração automática de hardware, objetivo três do experimento, foram realizados a partir da execução da aplicação. Utilizando componentes HDL para *atraso, multiplicação* e *somadores*, a arquitetura de filtro mostrada na Figura 6.6 foi implementada como base para este experimento.

A aplicação Embarcada, portanto, provê a entrada x[n] e lê a saída y[n] via software. Entradas simuladas foram usadas para os testes, dessa forma, é possível simular os valores de SNR que são necessários para execução da máquina de estados de gerenciamento mostrada na Figura 6.2.

Usando esse modelo de aplicação e filtro valores de SNR e limites de SNR foram testados, gerando diferentes ordens de filtro à medida que necessário. O tempo de reconfiguração e geração dos *bitstreams* relativos a alguns tamanhos de filtro testados são exibidos na Tabela 6.1.

O tempo de geração de uma nova arquitetura é relativamente alto, devido ao fato de depender dos algoritmos de roteamento envolvidos, já o tempo de reconfiguração é sempre constante devido ao fato de que, independentemente da ordem do filtro gerado, o tamanho, em bytes, do *bitstream* parcial é constante.

Figura 6.6 - Modelo de filtro digital implementado. Aumentar a ordem do filtro, nesse caso, significa aumentar a quantidade de *delays* e adicionar uma nova constante bn.



Fonte: Elaborada pelo Autor

Tabela 6.1 - Tempo de geração para diferentes arquiteturas de filtro.

Ordem do Filtro	Tempo de Geração	Tempo de Configuração
33	7:12 (min:seg)	545 ms
35	7:30 (min:seg)	545 ms
37	7:12 (min:seg)	545 ms
38	8:00 (min:seg)	545 ms

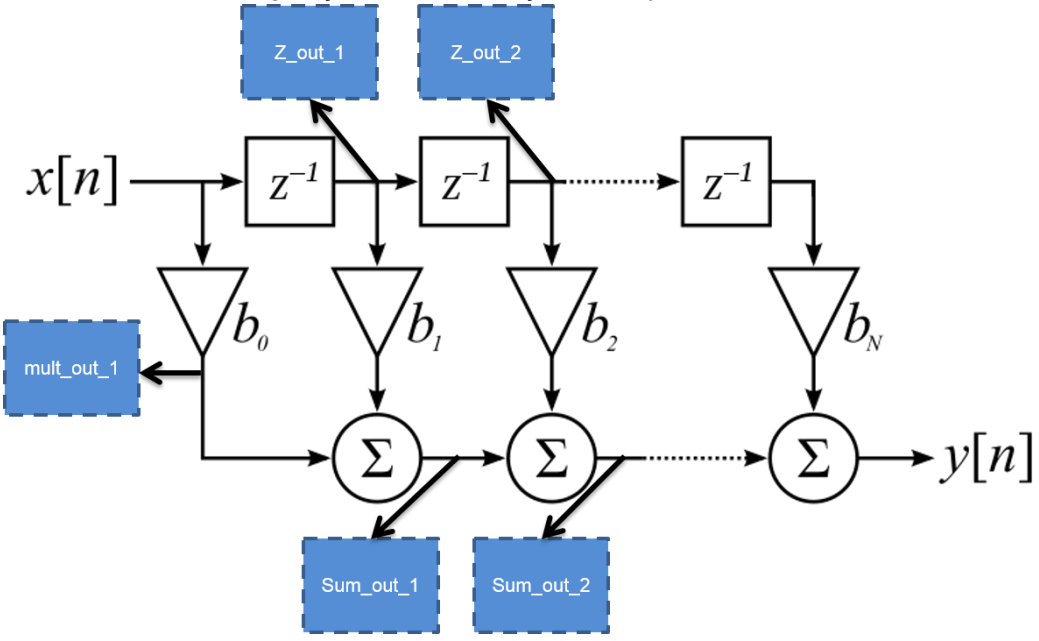
Fonte: Elaborada pelo Autor.

Através da tabela e da aplicação implementada, podemos constatar que o hardware foi gerado automaticamente pela aplicação e reconfigurado no sistema embarcado, atingindo assim os objetivos OS3 e OS4, esse último parcialmente.

Para alcançar os objetivos secundários OS4 e OS5 e testar o sistema de acesso aos componentes de hardware, algumas modificações foram necessárias na arquitetura da aplicação. No experimento, foram utilizadas as APIs de Comunicação do AHM para especificar que certas interfaces da Arquitetura de Hardware estariam acessíveis via software. Usando as APIs de geração para adicionar interfaces de comunicação, visando o melhor gerenciamento do hardware, podemos adicionar pontos de observação pelo caminho de dados do filtro digital, tornando-os visíveis em software quando necessário.

Para validar as APIs de comunicação e, ao mesmo tempo, testar seu impacto na geração das arquiteturas de hardware, foram adicionados pontos de comunicação em diferentes pontos do hardware da aplicação, como mostra a Figura 6.7.

Figura 6.7 - Exemplo de filtro com elementos de comunicação adicionados, exemplificando uma configuração de comunicação 2/1/2, para esse filtro.



Fonte: Elaborada pelo Autor.

A notação 2/1/2 denota que essa arquitetura de filtro contém dois elementos de comunicação para os dois primeiros registradores de atraso, um elemento de

comunicação para a saída do primeiro multiplicador e dois elementos de comunicação para as saídas dos dois primeiros somadores.

Para analisar os impactos da quantidade de elementos de comunicação no processo de geração de arquiteturas de hardware, foram criadas diferentes Arquiteturas Hardware para observação, para tamanhos variados de filtros. A Tabela 6.2 apresenta um sumário dos tempos para geração de arquiteturas e porcentagem da área reconfigurável utilizada pelos componentes de comunicação. Nesta tabela, a quantidade de elementos de comunicação utilizada na Arquitetura de Hardware é representada através da mesma notação utilizada na Figura 6.7. É possível observar que é significativa a quantidade de elementos de hardware necessária para realizar a observação nesse tipo de aplicação. Isso se dá pelo fato de a aplicação ser simples em termos de componentes e interconexões. Outro fato a ser observado é que o tempo de geração com e sem os componentes de observação não é tão diferente. O que mostra que as interfaces de observação não contribuem com muito overhead na geração de novas arquiteturas.

Tabela 6.2 - Resultados de tempo de geração e porcentagem da área usada pelos componentes de comunicação.

Ordem do Filtro	Componentes de Comunicação	% da área usada pelo hardware de comunicação	Tempo de Geração
33	2/1/1	1%	6:23
33	2/1/1	1%	6:40
34	2/2/1	1%	6:26
34	2/1/1	2%	7:00
35	2/2/2	2%	7:22

Fonte: Elaborada pelo Autor.

Levando em conta que as interfaces de acesso via software foram adicionadas de forma automática, e que o acesso via software das interfaces desejadas foi realizado com sucesso, utilizando as APIs providas pelo Repositório e Cliente AHM, consideramos o objetivo OS5 atingido.

Considerando ainda que as Arquiteturas de Hardware de teste necessárias para a execução dos experimentos foram geradas e configuradas na Plataforma de Hardware durante a execução da Aplicação Embarcada, usando as APIs providas pelo AHM, consideramos o objetivo OS4 atingido.

6.2 IMPLEMENTAÇÃO DE SISTEMAS AUTONÔMICOS

Uma vez testadas as funcionalidades básicas do AHM, passamos ao conjunto de testes para verificar se o repositório AHM consegue suportar a implementação de aplicações autonômicas. Uma vez que não foram encontradas na literatura arquiteturas de referência para sistemas autonômicos, fizemos uso da arquitetura básica proposta para este tipo de sistemas (HORN, 2001) (IBM, 2003) com vistas à implementação de aplicações com tais características.

Como visto na Seção 5.16, um sistema é considerado autonômico quando o mesmo se enquadra no modelo *Manage, Analyse, Plan, Execute – Knownledge* (MAPE- K). Naquela mesma seção, já mostramos como o AHM pode se enquadrar nesse modelo. No entanto, além do modelo de gerenciamento da aplicação, um sistema autonômico precisa ter algumas características básicas, já mencionadas anteriormente nos Capítulos 4 e 5, porém relembrado no Quadro 6.1. No escopo deste trabalho, consideraremos que a característica de Auto-Proteção e Auto-Reparo, presentes na referida Tabela, são opcionais. Uma vez que tais características não estão presentes em alguns sistemas disponíveis na literatura e considerados autonômicos.

No mapeamento que foi feito do AHM no modelo MAPE-K, e apresentado na Seção 5.16, citamos que alguns componentes e funcionalidades do modelo são dependentes de aplicação. Neste sentido, as aplicações de teste projetadas e apresentadas nas próximas subseções usam a arquitetura AHM orientada a contexto e implementam tais componentes e funcionalidades do MPAE-K.

Quadro 6.1 - Características dos sistemas autonômicos.

Característica	Descrição breve
Auto-Reparo	Sistemas Autonômicos possuem a habilidade de detectar e reparar falhas a fim de garantir a confiança no funcionamento do sistema.
Auto-Otimização	Sistemas autonômicos possuem a habilidade de realizar melhorias automaticamente a fim de manter alcançar os objetivos para qual foram projetados de forma mais eficiente.
Auto-Configuração	Sistemas autonômicos possuem habilidade de reflexão para realizar adaptação às variações dinâmicas no ambiente onde estão inseridos.
Auto-Proteção	Sistemas autonômicos possuem habilidade de prever e detectar ataques internos e externos, visando garantir a segurança.

Característica	Descrição breve
Auto-descrição	Sistemas autonômicos monitoram o que acontece internamente e externamente ao sistema.
Auto-Ajuste	Capacidade de identificar e se ajustar a situações adversas que possam acontecer com o sistema.
Auto-Sensibilidade / Sensibilidade ao contexto	Capacidade do sistema de sentir informações sobre o ambiente onde está inserido.

Fonte: Elaborada pelo Autor.

6.3 EXPERIMENTO 2: AUTONOMIC IMAGE SEGMENTATION SYSTEM (AISS)

Como já citado anteriormente no capítulo de cenários de uso, sistemas de processamento digital de imagens dependem de muitos fatores externos que podem influenciar diretamente no desempenho do mesmo. Por este motivo, sistemas deste tipo podem se beneficiar das possibilidades de adaptação ao contexto oferecidas pelo repositório AHM. Por outro lado, os sistemas de processamento digital de imagens podem também constituir um caso de testes para o repositório.

Um exemplo de aplicação na área, e que necessita de vários parâmetros para funcionar, é a segmentação digital de imagens. Para facilitar a leitura, a Figura 3.5 foi copiada para este capitulo e é reapresentada na Uma representação *das variáveis de contexto e seus possíveis valores pode ser construída visando fornecer ao sistema um meio de observa-las de forma automática.*

HS5 - *Regras de alto nível podem ser escritas através de relações condicionais entre as variáveis que descrevem o contexto e seus valores esperados, visando adaptar o sistema a possíveis condições refletidas em sua descrição de contexto.*

Figura 6.8. Esta figura apresenta um cenário simples, porém clássico, no qual câmeras de vídeo são instaladas em um cruzamento de trânsito e algoritmos são utilizados para fazer o processamento das imagens geradas pelas mesmas. Para simplificar o cenário, assumiremos que o algoritmo de segmentação utilizado é autocontido e suficiente para gerar as regiões segmentadas desejadas, sem o auxílio de outros algoritmos.

Uma vez apresentado o cenário de funcionamento no qual uma Aplicação de Segmentação de Imagens será utilizada como caso de teste para o AHM, o seguinte experimento foi elaborado:

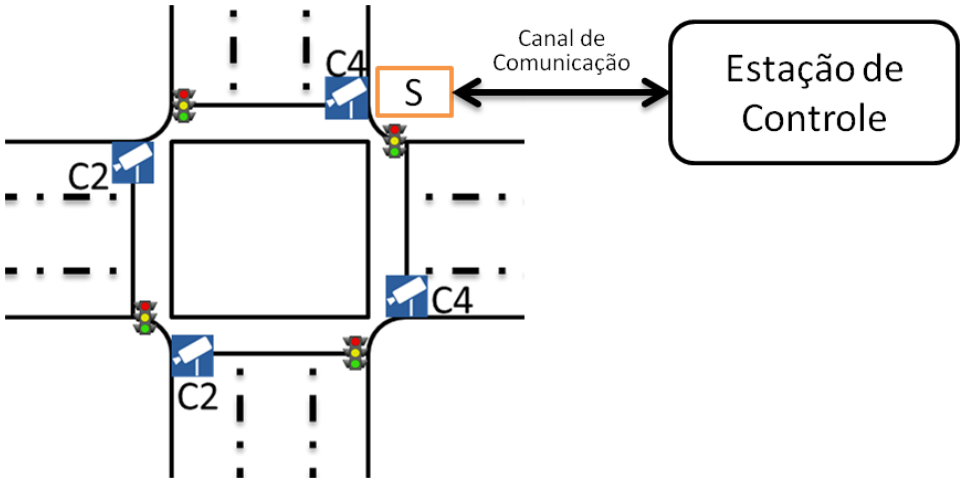
Experimento: Desenvolver uma aplicação autonômica para segmentação de imagens de cruzamento trânsito, usando o AHM orientado a contexto.

Objetivo Principal: Testar a validade das hipóteses secundárias de pesquisa HS5 e H6, apresentadas no Capítulo 1:

HS6 - *Uma representação das variáveis de contexto e seus possíveis valores pode ser construída visando fornecer ao sistema um meio de observa-las de forma automática.*

HS7 - *Regras de alto nível podem ser escritas através de relações condicionais entre as variáveis que descrevem o contexto e seus valores esperados, visando adaptar o sistema a possíveis condições refletidas em sua descrição de contexto.*

Figura 6.8 - Modelagem do problema para o *Autonomic Image Segmentation Filter.*



Fonte: Elaborada pelo Autor.

Para facilitar a consecução do objetivo principal deste experimento, os seguintes objetivos secundários são enumerados:

OS1- Modelar e implementar uma aplicação autonômica de segmentação de imagens, usando o modelo de repositório ativo orientado a contexto presente no AHM.

OS2- Implementar o mecanismo de representação do contexto da aplicação, através de variáveis, de acordo com o metamodelo usado no AHM.

OS3- Implementar os eventos de modificação de arquitetura, de acordo com o modelo de eventos de contexto presente no AHM.

Continuando com a modelagem da aplicação, os seguintes requisitos funcionais foram estabelecidos para a aplicação de testes *Autonomic Image Segmentation System*:

R1- A aplicação deverá implementar um sistema de segmentação de imagens, recebendo como entrada uma amostra de imagem e resultando em um conjunto de valores, denominados "clusters", representando as áreas segmentadas.

R2- A aplicação precisará implementar um mecanismo que a torne sensível às condições de iluminação das imagens processadas.

R3- A aplicação precisará escolher dentre os filtros implementados o que mais se adapte às condições de iluminação.

R4- A aplicação precisará ser sensível ao consumo de energia, sendo o sistema alimentado por bateria ou pela rede elétrica, economizando quando possível.

R5- A quantidade de clusters encontrados pelo processamento da imagem precisa se manter o mais constante possível.

A partir dos requisitos enumerados, foram extraídas e qualificadas as seguintes variáveis, que devem constituir o modelo de contexto do sistema:

Quadro 6.2 - Variáveis de contexto do sistema e seus possíveis valores.

Variável	Valores
FilterType	*string,* representando o nome do filtro usado. Ex: GaussianSmooth, LowPassSmooth
ThresholdValues	Vetor de Números Inteiros, com (3*n) elementos, onde n é o número de possíveis limiares de cor para cada pixel. Esses valores são modificados dependendo do formato de cor utilizado.
LighteningConditions	*float* (valor entre 0.0 e 1.0), que representa o valor da medição das condições de iluminação na imagem da câmera.

Variável	Valores
NumberOfClusters	*int*, número limite para a quantidade de clusters que deve ser mantida, considerando-se a condição de iluminação atual.
BatteryLevel	Inteiro (valor entre 0 e 255) que representa a carga da bateria.
EnergySupply	*bool* (true/false), que indica se o sistema está, ou não, sendo alimentado diretamente pela rede elétrica.
PowerProfile	Variável que pode assumir os valores constantes HIGH, MEDIOM ou LOW, de acordo com as seguintes condições: Se (EnergySupply = true), então PowerProfile=HIGH; senao se (130 <= BatteryLevel <= 255 and EnergySupply = false), então PowerProfile=MEDIUM; senao PowerProfile=LOW;

Fonte: Elaborada pelo Autor.

Para esse experimento, consideramos que as variáveis de contexto enumeradas no Quadro 6.2 são suficientes para a modelagem do contexto da aplicação. Para que o modelo de sensibilidade a contexto esteja completo, os eventos necessários às mudanças na Arquitetura de Hardware da Aplicação Embarcada precisam ser modelados. Contudo, tais eventos dependem da arquitetura de hardware adotada, uma vez que eles refletem mudanças na mesma.

A Figura 6.9 apresenta a modelagem dos componentes utilizados na implementação do AISS, incluindo os Componentes de Hardware presentes na Aplicação Embarcada. Neste caso, o gerenciamento da Arquitetura de Hardware do AISS é realizado pela Aplicação no Repositório, remotamente, com base nos eventos de contexto definidos.

Quadro 6.3 - Condições para Eventos de Contexto do AISS.

Evento de Contexto	Condições para Ocorrência do Evento
lighteningChangeThreshold1	LighteningConditions < 0.3 and LighteningConditions < 0.6 and NumberOfClusters < 4
lighteningChangeThreshold2	LighteningConditions < 0.3 and NumberOfClusters < 4

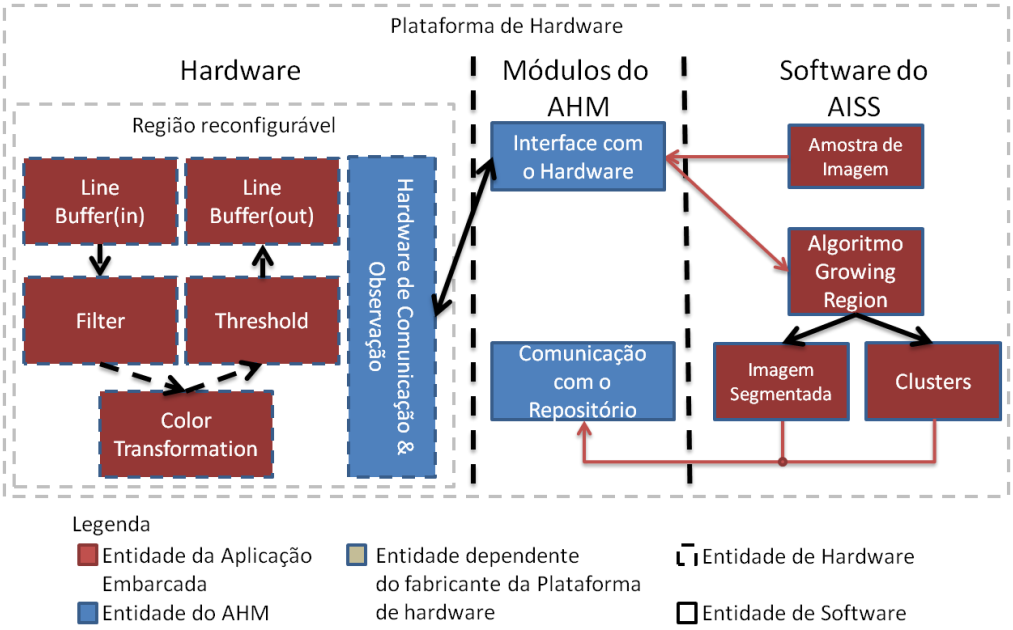
Evento de Contexto	Condições para Ocorrência do Evento
lighteningChangeThreshold3	LighteningConditions > 0.6 and NumberOfClusters < 4
powerProfileChanged1	PowerProfie = HIGH
powerProfileChanged2	PowerProfie = LOW
powerProfileChanged3	PowerProfie = MEDIUM

Fonte: Elaborada pelo Autor.

Finalmente, a Figura 6.10 apresenta a modelagem utilizada para implementação da Aplicação no Repositório para o AISS. Dessa forma, as condições de ocorrência dos Eventos de Contexto são mostradas na Quadro 6.3. Como pode ser observado nesta figura, a definição das variáveis e dos eventos de contexto é feita pelo gerente do sistema, enquanto as chamadas aos callbacks são automatizadas pelo AHM. As definições de eventos e variáveis de contexto para esse sistema, além de uma representação em pseudocódigo do callback utilizado na aplicação, podem ser vistos no Apêndice D.

Como já mencionado, o objetivo deste experimento consiste em implementar uma aplicação autonômica de segmentação de imagens. Para tanto, precisamos garantir as funcionalidades mínimas para esse tipo de sistema, como já recapitulado na Seção 6.2. É possível notar que, embora variáveis de contexto tenham sido definidas, bem como seus eventos de contexto, essas serão usadas para as tarefas se auto-gerenciamento do sistema autonômico. Uma funcionalidade ainda não explorada e que precisa ser provida, no âmbito dos sistemas autonômicos é a auto-otimização.

Figura 6.9 - Modelagem do *Autonomic Image Segmentation System* na Plataforma de Hardware.



Fonte: Elaborada pelo Autor.

Os componentes apresentados nas Figura 6.9 e Figura 6.10 implementam o AISS com a capacidade de se adaptar às condições de iluminação do ambiente. Contudo, para atender ao requisito R4, bem como prover a característica de auto-otimização dos sistemas autonômicos, o AISS implementa um componente denominado *Otimizador de Arquiteturas*, que é responsável por realizar a busca por uma Arquitetura de Hardware para a Aplicação Embarcada, que seja mais adequada à variável contextual *PowerProfile*.

Como já mencionado nas Seções 4.5 e 5.16, sistemas autonômicos remetem à necessidade de otimização de sua arquitetura durante a execução. Portanto, **para poder classificar o AISS como um sistema autonômico, a implementação de uma funcionalidade que aperfeiçoe o sistema durante sua execução é necessária**. Porém, devido à própria definição do conceito de otimização, esta não pode ser realizada sem a presença de informações que modelem o que precisa ser otimizado.

No caso do repositório AHM, a otimização da Arquitetura de Hardware da Aplicação Embarcada é feita com base nas informações sobre o contexto de execução desta aplicação e sobre os componentes armazenados no repositório.

Quadro 6.4 - Modelo usado para avaliar uma Arquitetura de Hardware na aplicação AISS.

$A = \{c_1, c_2, c_3, c_4, c_5\}$ $f(A) = \sum_{i=1}^{n} \text{Resource}(c_i)$ $g(A) = \sum_{i=1}^{n} Latency(c_i)$	*where* $f(A) \leq \text{AvailableResources}(R) - \text{controlUnit}(A)$ $c_j \in \text{N}^4 \wedge c_j = \{nRS, nLS, nBRAM, nDSP\}$ $n = carndinality(A)$

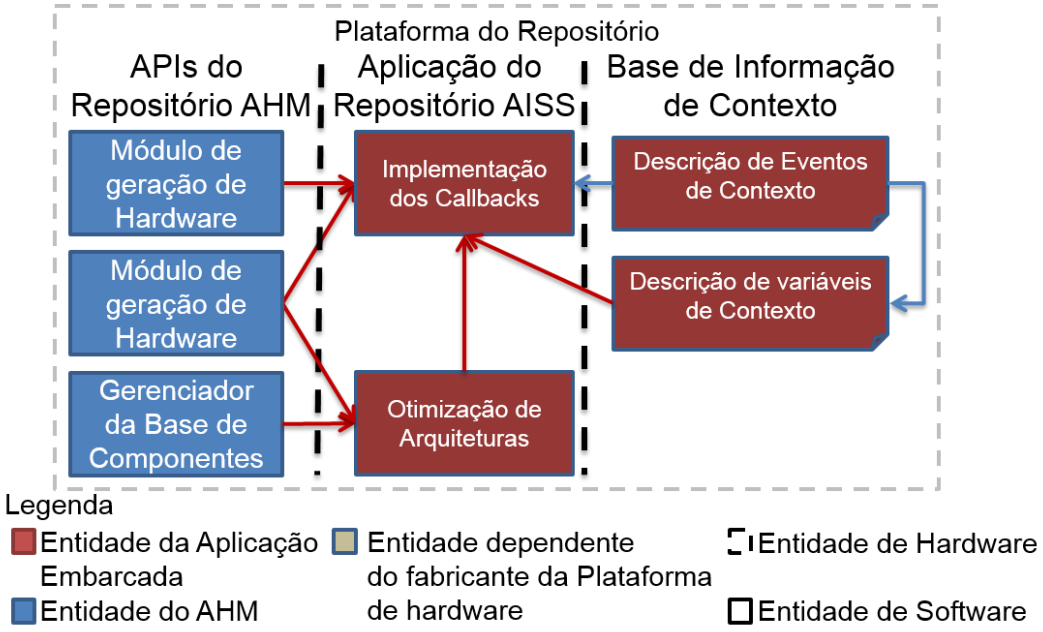
Fonte: Elaborada pelo Autor

O *Módulo Otimizador de Arquitetura* executa cada vez que um novo componente é adicionado no repositório ou enquanto uma arquitetura ótima para o contexto atual não foi encontrada. Para implementação desse módulo, foi desenvolvido um modelo para avaliação de uma dada arquitetura de hardware, tal modelo é descrito no Quadro 6.4.

No modelo de avaliação, uma arquitetura “A” é representada por um conjunto com cinco componentes representados por "cj", sendo dois deles do tipo *LineBuffers*, um do tipo *Filter*, um do tipo *Threshold* e um do tipo *Color Transformation*. Cada componente "*cj*" por sua vez é representado por um conjunto com 4 elementos: *nRS, nLS, nBRAM* e *nDSP* representando, respectivamente, a quantidade de *Register Slices, Logic Slices, Block RAM* e *DSP* usados pelo componente após sua síntese.

Ainda no modelo de avaliação, o elemento “*R*” representa a região reconfigurável na qual Arquitetura de Hardware será configurada. Esta região possui uma quantidade de recursos limitados, representada pela função *AvailableResources*(R). Essa quantidade precisa ser descontada da quantidade de recursos usados pelo componente de controle da Arquitetura (valor representado pela função *controlUnit*(A)), que é constante nesse experimento. O modelo ainda relaciona duas métricas usadas na aplicação: a quantidade de recursos usados por uma arquitetura (*Resource(A)*) e a latência da arquitetura (*Latency(A)*).

Figura 6.10 - Aplicação implementada no repositório AHM para o AISS.



Fonte: Elaborada pelo Autor

Para implementação da função *Latency*(A), um valor de latência é atribuído a cada componente presente no repositório. Este valor foi definido como sendo a quantidade de ciclos de relógio que o componente leva para fornecer uma saída válida após o fornecimento de uma nova entrada.

O valor da função *Resource*(A) por sua vez é calculado a partir das informações de síntese do componente, acessíveis através das APIs fornecidas pelo AHM. Mais especificamente, esta função deve resultar no somatório das quantidades de recursos (*Register Slices, Logic Slices, Block RAM* e *DSP*) utilizados por cada componente "*cj*" presente em "*A*".

A quantidade de recursos utilizada por cada componente de uma Arquitetura de Hardware é aproximadamente proporcional ao consumo de energia (XILINX CORPORATION, 2015) (ALTERA CORPORATION, 2012) do mesmo. Outra característica também observada é que o desempenho de um componente de hardware pode aumentar proporcionalmente à quantidade de recursos que o mesmo utiliza, devido ao maior emprego de paralelismo. No caso específico da aplicação AISS, assumiremos essa premissa como verdadeira, uma vez que todos os componentes de hardware que podem ser utilizados na sua Arquitetura de Hardware

foram desenvolvidos pelo mesmo programador. Essa premissa pode não ser válida para outras aplicações.

Figura 6.11: Equações de otimização das arquiteturas ótimas para o AISS.

$$A_{HIGH} = \{c_1, c_2, c_3, c_4, c_5\}$$

where

$$\mathrm{A_{HIGH}} = \mathrm{A} \mid \underset{\mathrm{A}}{\text{minimize}}\ g(A) - \frac{1}{\text{norma}(f(A))}$$

$$A_{LOW} = \{c_1, c_2, c_3, c_4, c_5\}$$

where

$$A_{LOW} = A \mid \underset{\mathrm{A}}{\text{minimize}}\ \text{norma}(f(A)) + \frac{1}{g(A)}$$

$$A_{MEDIUM} = \{c_1, c_2, c_3, c_4, c_5\}$$

where

$$\mathrm{A_{MEDIUM}} = \mathrm{A} \mid \underset{\mathrm{A}}{\text{minimize}}\ \text{norma}(f(A)) * (1 + \frac{1}{\mathrm{g(A)}})$$

where

$$f(A_{LOW}) \leq \text{ControlUnit}(A_{LOW}) + \text{AvailableResources}(R)$$

$$f(A_{HIGH}) \leq \text{ControlUnit}(A_{HIGH}) + \text{AvailableResources}(R)$$

$$f(A_{MEDIUM}) \leq \text{ControlUnit}(A_{MEDIUM}) + \text{AvailableResources}(R)$$

Fonte: Elaborada pelo Autor.

Desta forma, para satisfazer ao Requisito R4 da aplicação, referente à economia de energia, serão consideradas arquiteturas ótimas ou subótimas aquelas que utilizem a menor quantidade possível de recursos e que apresentem o melhor desempenho possível. Considerando-se tal definição, o problema de otimização da Arquitetura de Hardware do AISS é modelado através do conjunto de equações da Figura 6.11.

Na figura, os conjuntos A_{LOW}, A_{HIGH} e $\mathrm{A_{MEDIUM}}$ referem-se às arquiteturas que melhor satisfazem às condições expressas pela variável de contexto *PowerProfile*. O problema de encontrar todas as possíveis arquiteturas é do tipo multi-objetivo, onde precisamos encontrar mínimos e máximos de funções ao mesmo tempo.

A arquitetura $\mathrm{A_{HIGH}}$ é encontrada através da minimização do valor da função *g(A),* que se refere à latência da arquitetura, subtraído do inverso da *norma(f(A))*, que refere-se à utilização de recursos da Região Reconfigurável. Essa formulação

foi utilizada, pois, no caso em que duas arquiteturas possuam a mesma latência, escolhemos a arquitetura que possua o menor uso de recursos.

A arquitetura A_{LOW} é encontrada através da minimização do valor da norma da função *f(A)*, adicionado ao inverso do valor da função *g(A)*. Essa formulação nos permite escolher, no caso de empate, arquiteturas que possuam o menor uso de recursos com a menor latência possível.

Já a arquitetura A_{MEDIUM} é encontrada através da minimização do valor da função *norma(f(A))* multiplicada por *1 + 1 / g(A)*. Semelhantemente ao problema de encontrar a arquitetura A_{LOW}, essa formulação permite escolher uma arquitetura que minimize a utilização de recursos com o melhor desempenho possível. No entanto, esta formulação atribui um peso maior ao termo da função de avaliação que modela a latência da Arquitetura, permitindo arquiteturas com maior desempenho do que no caso de A_{LOW}.

6.3.1 Execução do Sistema

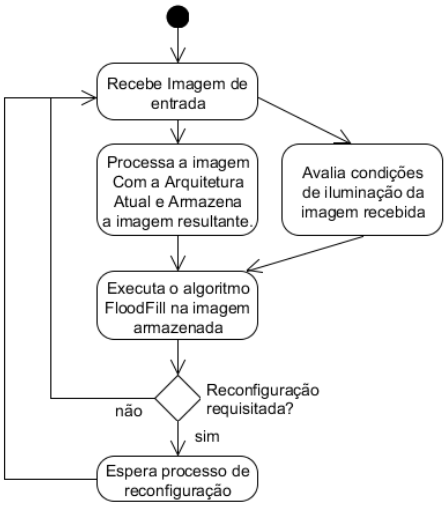
Uma vez projetado o sistema, a implementação foi realizada em uma plataforma Xilinx Virtex 5 XUPV5 (XILINX CORPORATION, 2011). O repositório foi implementado em um desktop Intel Core I7 3.4 GHz, com 8 GB de memória RAM executando Linux através de uma máquina virtual. O sistema foi sintetizado usando o modelo de desenvolvimento orientado à plataforma.

A aplicação inicia com a configuração inicial utilizando *Line Buffer Serial* tamanho 3, *Image FilterLow Pass Serial* tamanho 3, *Serial Color Transformation*, *Serial Threshold* tamanho 3. Nesse caso o *Image Filter Low pass* apenas reduz o ruído na imagem de entrada, evitando grandes áreas borradas. Não são realizadas transformação de cor, portanto para essa configuração o componente *Color Transformation* não realiza tarefas. O componente *Threshold* contem apenas três conjuntos de cores, portanto irá descartar os demais conjuntos, forçando para Preto (0, 0, 0 em RGB) os pixels que não pertençam a algum dos conjuntos escolhidos.

Imagens base, de calibração, foram escolhidas para cada uma das condições de iluminação levadas em consideração no Quadro 6.3, após isso a aplicação processará imagens aleatórias seguindo um padrão fixo de condições de iluminação, indo de iluminações consideradas normais (*LighteningConditions* > 0.6*)*

seguindo para condições mais escuras (*LighteningConditions* < 0.6 ou *LighteningConditions* < 0.3).

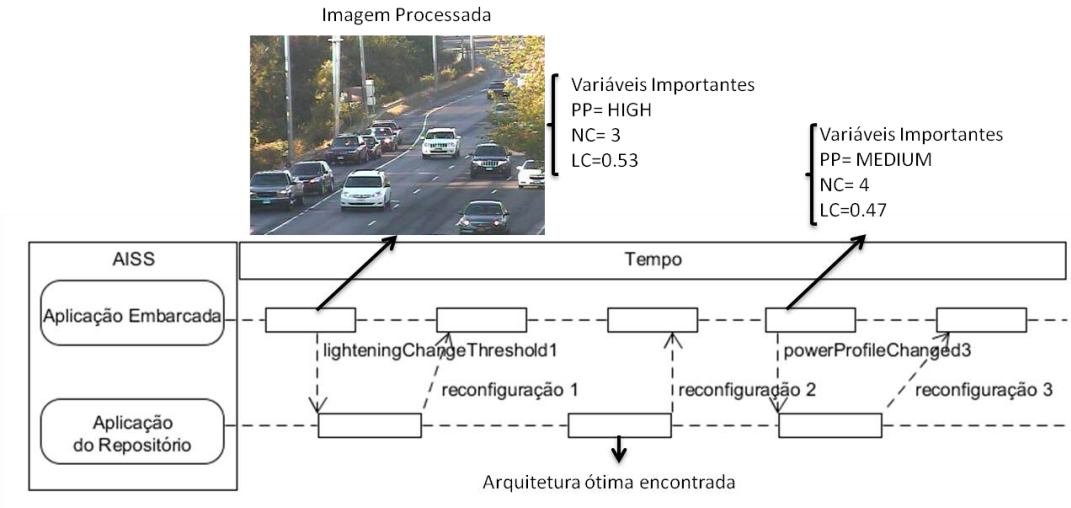
Figura 6.12 - Execução da Aplicação Embarcada do AISS



Fonte: Elaborada pelo Autor.

A Figura 6.12 mostra como é feita a execução da aplicação no cliente embarcado, uma imagem é mandada via ethernet pela Aplicação do Repositório e recebida pelo Cliente AISS. A imagem é processada pela arquitetura de hardware atual, e suas condições de iluminação são calculadas através da medida da média do valor Y da imagem convertida para o formato *YCbCr*.

Figura 6.13 - Ilustração de alguns eventos de reconfiguração que podem ocorrer durante a execução da aplicação AISS. PP denota a variável *PowerProfile*; NC denota a variável *NumberOfClusters*; LC denota a variável *LighteningConditions.*



Fonte: Elaborada pelo Autor.

A Figura 6.13 ilustra alguns eventos de reconfiguração que podem ocorrer durante a execução do AISS. Como dito, imagens são fornecidas e avaliadas para identificar as condições de iluminação, quando uma condição de iluminação em certo intervalo é identificada um evento pode ser disparado, como no caso do evento lighteningChangeThreshold1.

Outro tipo de reconfiguração pode ocorrer caso uma arquitetura ótima seja encontrada e gerada pelo repositório. Isso ocorre no caso da reconfiguração 2. Por ultimo a Reconfiguração 3 ocorre pois o evento powerProfileChanged2 ocorreu, indicando que houve alguma modificação no sistema de fornecimento de energia da aplicação embarcada.

Ambos os eventos e reconfigurações mostrados na Figura 6.13 são artificiais, ocorrendo de acordo com um algoritmo de geração aleatória. Porém, para esse experimento, consideramos a geração aleatória de eventos suficiente para os testes que seguem.

6.3.2 Análise dos Resultados

Para avaliar as funcionalidades autonômicas e verificar se o sistema satisfaz às características requeridas para ser classificado como um Sistema Autonômico, nas subseções que seguem iremos comentar cada uma das características mostrando como o AISS as implementa.

6.3.3 Auto-Otimização

Conforme mencionado anteriormente, a otimização do sistema é executada utilizando o modelo apresentado através das equações do Quadro 6.4 e Figura 6.11. Para o sistema de auto-otimização, foram implementadas diferentes versões, mostradas na Tabela 6.3, de cada componente presentes na Arquitetura de Hardware. Nesta tabela, os valores de *nThresholdValues* e *filterSize*, representam o número de limiares de cor no componente *Threshold* e os tamanhos dos filtros de imagem usados para o componente *Filter*. Esses valores são conhecidos de antemão, sendo iguais a 3 e 6, nesse caso.

A função principal do *Módulo Otimizador* da AISS é encontrar a configuração ótima de componentes para a Arquitetura de Hardware da Aplicação Embarcada, de acordo com a variável de contexto *PowerProfile*. Em conjunto com as APIs do AHM e o modelo de *Eventos de Contexto* usado, o *Módulo Otimizador* é capaz de encontrar os componentes ótimos para os três valores possíveis da variável *PowerProfile (HIGH, MEDIUM e LOW)*.

Desta forma, concluímos que a funcionalidade de Auto-Otimização foi satisfatoriamente implementada pelo AISS.

Tabela 6.3 - Versões dos componentes implementados para teste do modelo de otimização.

Componente	Resource(c)				Latency(c)
	nRS	*nLS*	*nBRAM*	*nDSP*	
Line Buffer Serial	9	9	1	0	1
Line Buffer Parallel	3	6	2	0	1
Filter LowPass Serial Iterative	210	651	0	0	2*(filterSize * filterSize + 1)
Filter LowPass Parallel Iterative	237	789	0	0	3*(filterSize + 1)
Filter LowPass Parallel Direct	284	351	0	0	filterSize
Filter LowPass Serial Direct	144	316	0	0	filterSize*filterSize
Serial Color Transformation	21	37	0	0	12
Parallel Color Transformation	36	92	0	0	4
Serial Threshold	25	92	0	0	nThresholdValues
Parallel Threshold	42	112	0	0	nThresholdValues

Fonte: Elaborada pelo Autor

6.3.4 Auto-Configuração e Auto-Ajuste

Usando a arquitetura proposta no AHM, baseada na interação entre cliente e repositório ativo de componentes, foi possível ajustar ou reconfigurar a Arquitetura de Hardware da Aplicação Embarcada do AISS. Adicionalmente, a implementação do mecanismo de representação contextual e de eventos de contexto provê uma forma automática, e de alto nível, de configurar ou ajustar o sistema.

Como já mostrado no Experimento 1, os tempos de geração para determinadas arquiteturas podem variar, enquanto o tempo de reconfiguração é constante. Para esse experimento, o tempo de reconfiguração permaneceu fixo em 500ms. A

Tabela 6.4 apresenta alguns tempos de geração para diferentes arquiteturas de hardware que foram testadas nessa aplicação. Através desta Tabela, podemos perceber a capacidade do sistema de gerar diferentes arquiteturas.

Tabela 6.4 - Tempo de geração para algumas arquiteturas.

Arquitetura	Tempo de Geração
Arquitetura A	5:37 min
Arquitetura B	6:27 min
Arquitetura C	5:49 min

Fonte: Elaborada pelo Autor.

As Arquiteturas de Hardware da aplicação AISS utilizam os seguintes Componentes de Hardware: Arquitetura A utilizando *Line Buffer Serial* tamanho 3, *Image FilterLow Pass Serial* tamanho 3, *Serial Color Transformation*, *Serial Threshold* tamanho 3; Arquitetura B contendo *Line Buffer Parallel* tamanho 5, *Image FilterLow Pass Iterative* tamanho 5, *Parallel Color Transformation*, *Serial Threshold* size 3; Arquitetura C utilizando *Line Buffer Serial* tamanho 3, *Image FilterLow Pass Serial* tamanho 3, *Serial Color Transformation*, *Serial Threshold* tamanho 10;

Os resultados da

Tabela 6.4 e o tempo necessário para a reconfiguração da aplicação, corroboram com os resultados obtidos no Experimento 1. As arquiteturas A, B e C mostradas, representam as arquiteturas levadas em consideração para montar a Figura 6.13, junto da figura, os tempos de geração e configuração demonstram as capacidades de auto-configuração e auto-ajuste da aplicação AISS.

6.3.5 Auto-Descrição

A característica de autodescrição foi explorada no Experimento 1, utilizando a aplicação *Hardware Reconfigurable Filter*, no qual os componentes da arquitetura foram descritos usando o modelo de componentes proposto pelo AHM. Já neste experimento com o sistema AISS, o foco foi dado à validação da modelagem das variáveis de contexto que afetam o sistema, bem como dos eventos de contexto importantes ao funcionamento do mesmo. Para tanto, o APÊNDICE D apresenta os detalhes de um mecanismo de representação contextual para o AISS. Tal mecanismo contempla os eventos de contexto descritos no Quadro 6.3 e as

variáveis de contexto da Quadro 6.2, que conjuntamente modelam o contexto de operação do AISS.

Portanto, adicionando-se aos resultados obtidos no Experimento 1 os resultados obtidos neste Experimento, consideramos que a capacidade de autodescrição do AISS, utilizando o repositório AHM, foi satisfeita.

6.3.6 Auto-Sensibilidade / Sensibilidade ao contexto

A característica de Auto-sensibilidade diz respeito à capacidade do sistema de perceber o seu estado corrente. Para o AISS, desenvolvemos um componente Observador de Hardware, previsto no Repositório AHM, capaz de monitorar a quantidade de clusters na Aplicação e representar tal informação na as variáveis de contexto descritas no Quadro 6.2.

Tabela 6.5 - Custo da adição do hardware de observação nos componentes, para garantir a característica de sensibilidade ao contexto. As arquiteturas usadas foram as mesmas mostradas na Tabela 6.4

Arquitetura	Recursos usados	Porcentagem de aumento de área
Arquitetura A	160 RLUT	5.0%
Arquitetura B	230 RLUT	6.9%
Arquitetura C	172 RLUT	3.2%

Fonte: Elaborada pelo Autor.

Para avaliar o impacto da adição da característica de sensibilidade a contexto nos componentes de hardware do sistema, apresentamos na Tabela 6.5 o aumento na área de FPGA, proporcionado pela adição dos componentes de observação, em algumas Arquiteturas de Hardware testadas para o AISS.

O componente Observador de Hardware do AISS foi capaz de monitorar o estado atual da aplicação e, além disso, gerar eventos de contexto para notificar sobre alterações neste estado. Desta forma, consideramos que a característica de auto-sensibillidade está presente no sistema AISS, implementado de acordo com o modelo proposto neste trabalho.

6.3.7 Avaliação do AISS

Considerando-se os resultados apresentados nas Seções 6.3.3 à 6.3.6, podemos observar que o AISS provê cinco das sete características principais de uma aplicação autonômica. Neste caso, as funcionalidades de auto-reparo e autoproteção não foram implementadas.

A funcionalidade de auto-reparo, diz respeito à capacidade da aplicação identificar e corrigir falhas, que venham a ocorrer durante a execução. Essa capacidade poderia ser implementada através de um sistema de detecção de falhas que executasse junto à Aplicação Embarcada. Tal sistema poderia consistir tanto de componentes de Hardware quanto de Software, com o intuito de identificar as falhas ocorridas no Hardware da Aplicação.

Uma vez identificadas as falhas, a funcionalidade de autoconfiguração, ou auto-ajuste poderiam ser usadas para realizar o reparo do Hardware a fim de corrigi-las. No entanto, a simples detecção de falhas em Sistemas de Hardware requer muito esforço de desenvolvimento, consistindo por si, uma Grande Área de estudo no campo de desenvolvimento e validação de Arquiteturas de Hardware. Por esse motivo, resolvemos deixar implementações a respeito da funcionalidade de Auto-reparo para trabalhos futuros.

A autoproteção diz respeito à capacidade da aplicação de antever problemas e evitá-los. Esta característica poderia ser implementada através de um sistema de testes que verificasse o comportamento da aplicação para qualquer situação possível, incluindo qualquer sequência de estímulos de entrada. Neste caso, as arquiteturas geradas pelo AHM poderiam ser validadas no sistema de testes antes de ser enviadas à aplicação para reconfiguração.

As ferramentas providas pelos fabricantes de plataformas de Hardware Reconfigurável contemplam sistemas de testes que poderiam ser utilizados para, de forma automática, validar as arquiteturas geradas pelo AHM. Contudo, a realização de tais testes requereria o projeto de uma suíte de testes para aplicação que, para ser completa, iria demandar um grande esforço. Além disso, tal suíte seria específica para a aplicação em questão. Por este motivo, a demonstração da característica de autoproteção foi deixada fora do escopo deste trabalho, ficando também para trabalhos futuros.

Outros sistemas considerados autonômicos na literatura também não implementam mecanismos de autoproteção e auto-reparo. Comparando-se as características destes sistemas com as do AISS, no escopo deste trabalho, consideraremos que o AISS, implementado de acordo com a metodologia proposta pelo repositório AHM, é um sistema autonômico. Além disso, a partir dos resultados dos experimentos realizados, consideramos que os objetivos específicos O1, O2 e O3 desta Tese foram alcançados.

6.4 EXPERIMENTO 3: AUTONOMIC AUTO PILOT (AAP)

As aplicações apresentadas nas Seções 6.1 e 6.3 desenvolvidas a partir do rascunho, sem utilizar como base qualquer implementação preexistente, visando comprovar algumas características do repositório AHM. Com o objetivo de avaliar o esforço de adaptar uma aplicação existente, de forma a torna-la autonômica, através da integração com o repositório AHM, nesta seção apresentamos os experimentos realizados com a aplicação Autonomic Auto Pilot (AAP).

A Autonomic Auto Pilot (AAP) consiste em uma aplicação de piloto automático para veículos aéreos não tripulados. Originalmente, a aplicação fornecida pela empresa 3D Robotics consistia em um software embarcado em um microcontrolador, com entradas para os diversos sensores presentes em uma aeronave e saídas para os diversos atuadores também presentes na mesma.

Neste trabalho, a arquitetura do software original da aplicação AAP foi organizada em módulos, de acordo com as suas funcionalidades. Alguns destes módulos foram reimplementados em HDL e sintetizados em hardware. Outros módulos foram mantidos em software. Em seguida, a aplicação foi integrada com o repositório AHM para possibilitar a sua adaptação aos diferentes contextos de operação, tornando-a autonômica. Desta forma, podemos definir:

Experimento: Implementar um sistema autonômico a partir de um sistema não autonômico preexistente, usando arquitetura AHM sensível a contexto.

Objetivo Principal: Reafirmar as capacidades e flexibilidade da solução, analisar os impactos dos componentes de software da arquitetura no funcionamento de um sistema embarcado de referência.

Os seguintes objetivos secundários são listados para esse experimento:

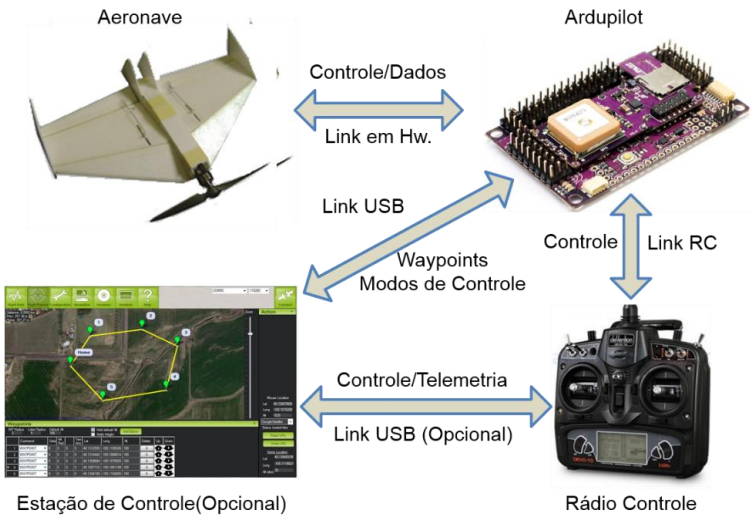
OS1- Projetar e implementar uma aplicação autonômica derivada de um sistema embarcado de referência a ser escolhido.

OS2- Mostrar que é possível incorporar o AHM e seu modelo de funcionamento em um sistema de referência.

OS3- Verificar o impacto dos componentes derivados do AHM na arquitetura e funcionamento do sistema embarcado de referência escolhido.

Citamos na seção de trabalhos relacionados que é possível implementar uma aplicação autonômica para um VANT. Essa aplicação substituiria alguns componentes de hardware em caso de falha, podendo realizar reparos durante o voo. No entanto uma aplicação desse porte iria requerer que as vias de dados, que entram e saem do componente substituído, fossem replicadas na área reconfigurável, o que, no escopo desta Tese não é realizável. Porém, nesse mesmo contexto de Aplicação, podemos modificar o software de piloto automático usado, o Ardupilot (C., 2010), para implementar uma aplicação autonômica, servindo, portanto, como nossa aplicação de referência para comparações.

Figura 6.14 - Componentes presentes no modelo comum de funcionamento do Ardupilot.



Fonte: Elaborada pelo Autor.

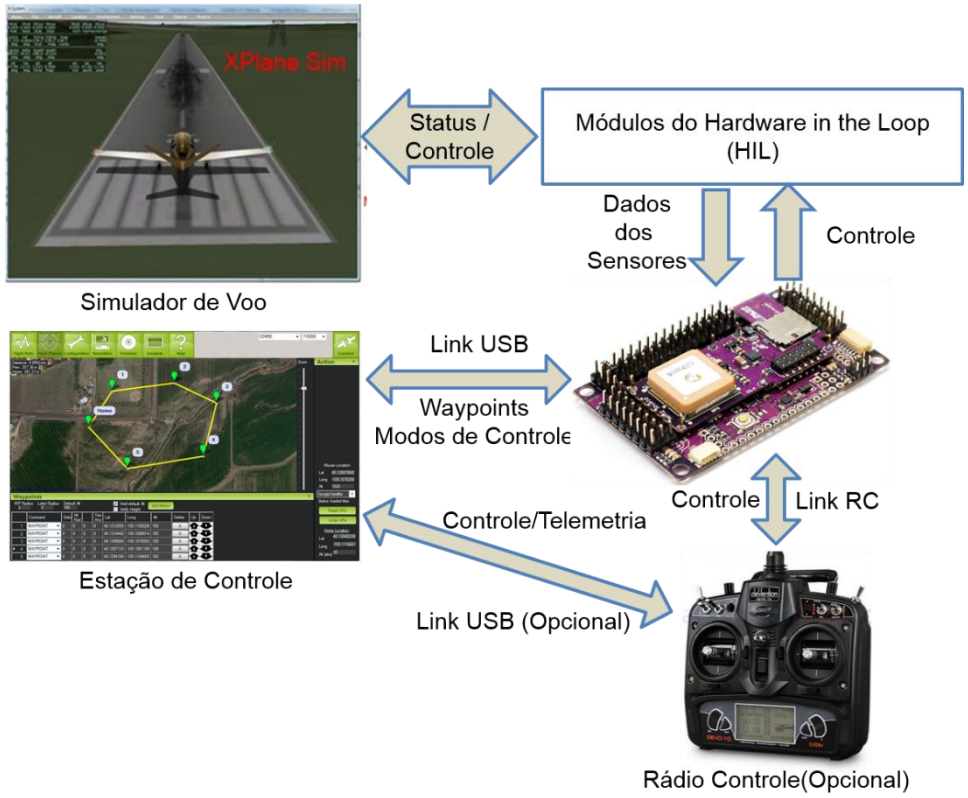
A Figura 6.14 ilustra os elementos presentes no modelo de funcionamento padrão quando implantamos o Ardupilot em uma aeronave. Como pode ser visto, a presença de uma estação de controle é opcional durante o funcionamento, bem como sua ligação com o equipamento de rádio controle. Porém, é necessário que o

controlador (Ardupilot) seja conectado ao menos uma vez à estação de controle, para que os *waypoints* correspondentes à missão planejada sejam carregados.

Um modo alternativo de funcionamento possível para o Ardupilot é o modo *Hardware in the Loop* (HIL). Esse modo é usado para testar as missões e algoritmos de controle, antes que sejam testados na aeronave real. Os elementos ilustrados, anteriormente, ainda podem estar presentes nesse modo de operação, porém, ao invés de controlar um aeromodelo real, o Ardupilot irá controlar uma aeronave em um ambiente simulado.

No modo de funcionamento HIL, ilustrado na Figura 6.15, apenas algoritmos presentes no hardware são utilizados, enquanto os valores de sensores são recuperados através da interação com o sistema de simulação.

Figura 6.15 - Elementos presentes no modo de operação *Hardware in the Loop* para o Ardupilot.



Fonte: Elaborada pelo Autor.

O modo em HIL é mais amigável para testes dos algoritmos de controle, uma vez que não põe em risco a aeronave real. O modo HIL suporta ainda a adição do módulo de rádio controle, caso desejado. Para implementação da aplicação do Experimento, portanto, resolvemos usar o modo HIL, modificando o software que executa no controlador para que pudesse ser executado em uma plataforma com

acesso a um dispositivo reconfigurável. Usando esse modelo de implantação, os requisitos da aplicação para o experimento encontram-se listados abaixo:

R1- A aplicação deverá implementar os componentes de controle de navegação em hardware, deixando o controle de navegação tipo *Loiter* implementado em software por motivos de segurança.

R2- A aplicação deverá modificar o hardware, se necessário, quando diferentes tipos de controle de navegação forem selecionados.

R3- A aplicação deverá se adaptar à quantidade de energia disponível, reduzindo o consumo à medida que necessário.

Com os requisitos e avaliando os componentes de controle automático do Ardupilot, podemos enumerar um conjunto de variáveis de contexto apresentadas no Quadro 6.5, enquanto as condições para os Eventos de Contexto são mostradas no Quadro 6.6:

Quadro 6.5 - Variáveis de contexto modeladas para o Experimento 3.

Variável	Valores
NavigationType	Inteiro (valor entre 0 e 2), representando um dos três tipos de navegação *Loiter, Waypoint, HeadingHold*.
BatteryLevel	Inteiro (valor entre 0-255), simula o nível de carga da bateria.
PowerProfile	Variável que pode assumir os valores constantes *HIGH*, *MEDIUM* ou *LOW*, de acordo com as seguintes condições: Se (*BatteryLevel* > 190), então *PowerProfile=HIGH*; Se (100 <= *BatteryLevel* <= 190), então *PowerProfile=MEDIUM*; Se (100 < *BatteryLevel*), então *PowerProfile=LOW*;

Fonte: Elaborada pelo Autor.

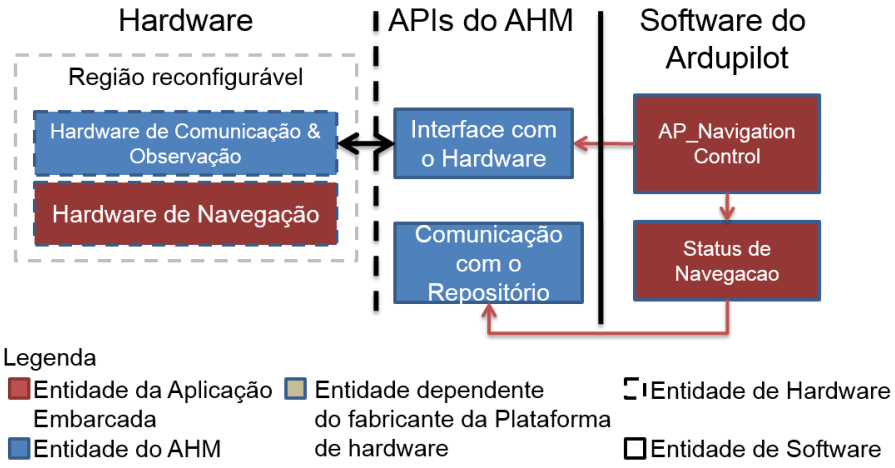
Quadro 6.6 - Condições para os Eventos de Contexto de Contexto modelados para o Experimento 3.

Evento	Condições para Ocorrência
navigationTypeChanged1	*NavigationType = Loiter*
navigationTypeChanged2	*NavigationType = Waypoint*
navigationTypeChanged3	*NavigationType = HeadingHold*
powerProfileChanged1	*PowerProfie = HIGH*
powerProfileChanged2	*PowerProfie = LOW*
powerProfileChanged3	*PowerProfie = MEDIUM*

Fonte: Elaborada pelo Autor.

As variáveis e eventos nos Quadros Quadro 6.5 e Quadro 6.6 foram retirados dos requisitos e dos algoritmos de navegação que fazem parte do software embarcado presente no Ardupilot. Com essas variáveis e requisitos em mente é possível modelar os componentes embarcados do sistema, como mostrado na Figura 6.16:

Figura 6.16 - Componentes da aplicação embarcada do Experimento 3.



Fonte: Elaborada pelo Autor.

A Aplicação Embarcada foi implementada com a adição dos módulos do AHM no módulo *AP_Navigation Control* do Ardupilot. Esse módulo passará a usar as APIs do AHM para acessar os controladores implementados em hardware, fazendo a navegação em software quando a mesma for do tipo *Loiter*. Esse módulo também fará as leituras e estimativas do status de navegação e nível de bateria para poder fornecer ao repositório informações sobre as variáveis de contexto *NavigationType* e *BatteryLevel*.

Em seguida, modelamos a Aplicação no Repositório que irá realizar o gerenciamento junto ao Repositório AHM, assim como no Experimento 2. A implementação da aplicação no repositório cobre as descrições das variáveis e eventos de contexto seguindo o meta-modelo adotado pelo AHM, as implementações dos callbacks a serem executados para tratamento dos eventos e um módulo de otimização de Arquiteturas de Hardware.

Devido à complexidade e especificidade das Arquiteturas de Hardware que implementam os algoritmos de navegação, o modelo de otimização irá focar na

redução dos recursos utilizados pelos Componentes de Hardware em termos ajustes e configurações internas.

Através das implementações em das Arquiteturas de Hardware que implementam os algoritmos de controle *Waypoint* e *HeadingHold*, foi observado que a arquitetura de componentes de cada controlador é, completamente diferente uma da outra. Por isso dependendo as equações que avaliam os componentes, apresentadas no Experimento 2, precisam ser modificadas da seguinte forma:

Quadro 6.7 - Modificações feitas nas equações do Quadro 6.4, para poderem ser utilizadas no Experimento 3.

$A_{\text{Waypoint}} = \{c_{wp1}, c_{wp2}, c_{wp3}, c_{wp4}\}$ $A_{\text{HeadingHold}} = \{c_{hh1}, c_{hh2}, c_{hh3}\}$ $f(A) = \sum_{i=1}^{n} \text{Resource}(c_i)$ $g(A) = \sum_{i=1}^{n} Latency(c_i)$	$where$ $f(A) \leq \text{AvailableResources}(R) - \text{controlUnit}(A)$ $c_j \in \text{N}^4 \wedge c_j = \{nRS, nLS, nBRAM, nDSP\}$ $n = carndinality(A)$

Fonte: Elaborada pelo Autor.

Como pôde ser visto nas equações do Quadro 6.7, os componentes para uma dada arquitetura precisam ser escolhidos de um diferente conjunto dependendo do tipo de controlador ativo. Os Componentes de Hardware otimizáveis nas arquiteturas são: para o controlador de navegação *Waypoint*, dois módulos *Cordic* (XILINX CORPORATION, 2010), um módulo multiplicador de ponto flutuante, um módulo de soma de ponto flutuante; para o controlador de navegação *HeadingHold* um módulo *Cordic*, um módulo de soma de ponto flutuante e um módulo de multiplicação de ponto flutuante.

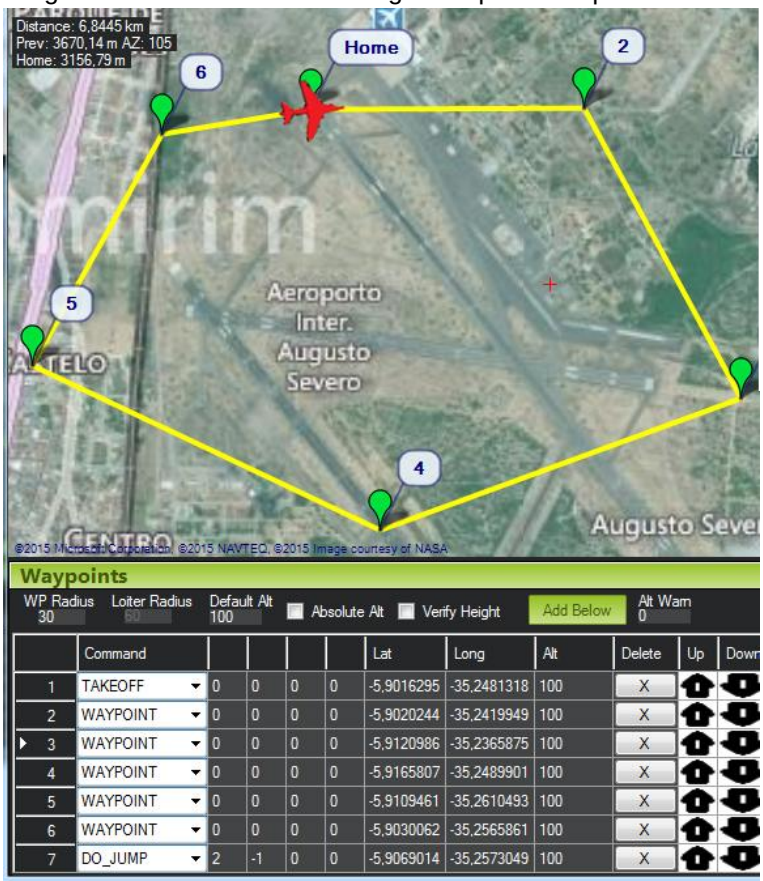
Nesse caso, o valor representado pela função *controlUnit(A)*, varia também de acordo com o tipo de controlador ativo. Nos cálculos das arquiteturas ótimas, as mesmas funções objetivas apresentadas no conjunto de equações mostrado na Figura 6.11 serão usadas.

6.4.1 Execução do Sistema

A implementação do sistema foi feita usando o modo de operação HIL, seguindo o esquema apresentado na Figura 6.19. Como pode ser visto, o controlador de hardware foi substituído por uma Plataforma reconfigurável ZedBoard e a comunicação envolvida no modo de operação HIL foi implementada usando ethernet. A Plataforma do Repositório é um Desktop com processador Intel Core I3, 3.0 GHz, 6.00 GB de memória RAM e executando Linux. O sistema foi sintetizado usando o modelo de desenvolvimento orientado à plataforma, assim como nos demais experimentos.

A aplicação executa usando o modelo HIL, apresentado na seção anterior, inicialmente uma rota é programada para ser seguida pelo piloto automático, a rota configurada é mostrada na Figura 6.17. Na configuração mostrada, a aeronave seguirá pelos pontos marcados até chegar no ponto 6, onde irá mudar de rota seguindo para o ponto 2, indefinidamente.

Figura 6.17 - Rota de voo configurada para o Experimento 3.



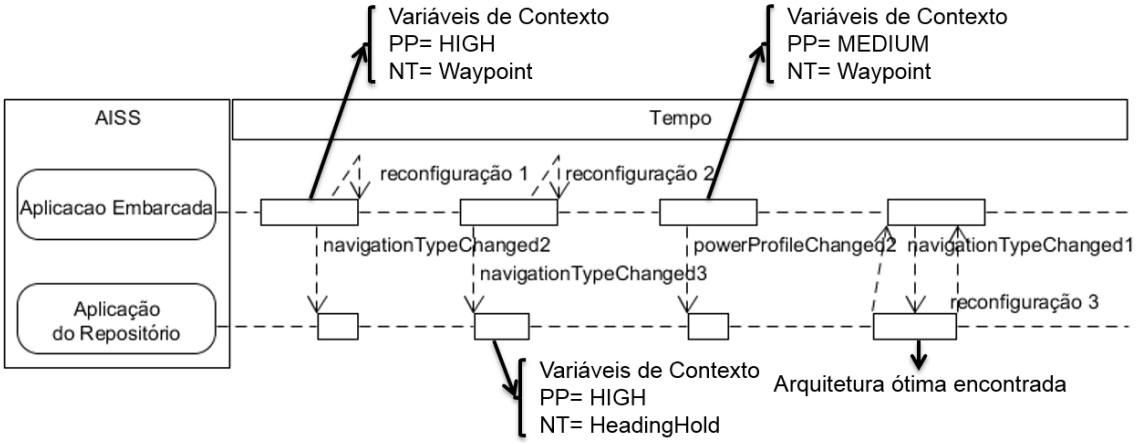
Fonte: Retirada do Programa *Mission Planner* (3DROBOTICS).

Tendo que a rota mostrada na figura executa indefinidamente, é possível testar os componentes e arquiteturas de hardware implementadas para esse experimento configurando mudanças no modo de voo a partir da posição da aeronave na rota. Dessa forma a aeronave foi programada para usar o modo de voo *HeadingHold*, sempre que passa pelos pontos 2 e 4, voltando para o modo *WayPoint* após 4 minutos de voo.

O simulador *X-Plane* (LAMINAR RESEARCH) possui um sistema de simulação de bateria/combustível que pode ser configurado para fins de simulações mais acuradas. Através desse sistema configuramos a aeronave para suportar 30 minutos de voo, após esse tempo a simulação precisa ser reiniciada.

O Hardware Controlador inicia com duas arquiteturas carregadas no sistema de arquivos local, uma para o modo *Waypoint* e outra para o modo *HeadingHold*. Após a decolagem o modo de controle *Waypoint* é ativado até que os pontos de mudança de modo de controle sejam alcançados.

Figura 6.18 - Sequência de reconfigurações e mudanças no contexto encontradas no Experimento 3



Fonte: Elaborada pelo Autor.

A Figura 6.18 mostra alguns eventos em sequência que foram registrados durante a execução do experimento. Inicialmente o AAP se reconfigura modificando o modo de piloto automático para *Waypoint*, depois disso, ao chegar ao ponto 2, muda novamente seu modo de piloto para *HeadingHold*. As mudanças no contexto são informadas à Aplicação do Repositório através dos eventos de contexto. Após alguns minutos de execução a variável *PowerProfile* muda seu valor para *MEDIUM*.

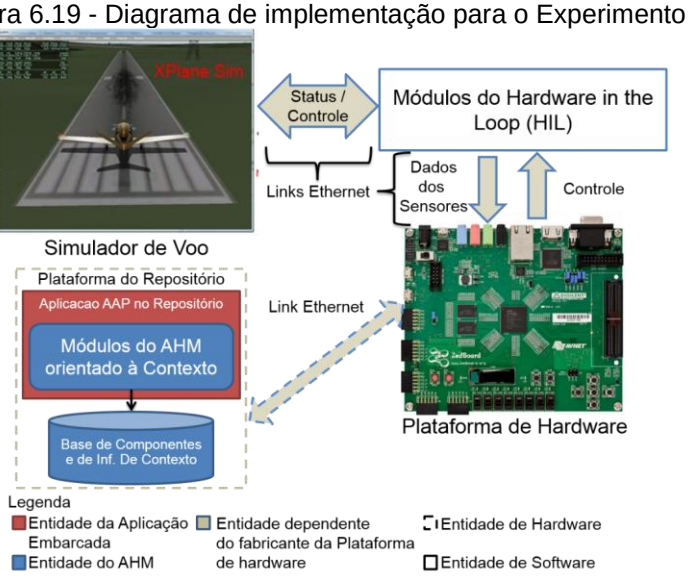
Quando uma arquitetura ótima é encontrada e gerada o repositório avisa a aplicação de piloto automático que, para fazer o download das novas arquiteturas de hardware muda o modo de piloto automático para *Loiter*, que é implementado em software. Após o download da arquitetura a aplicação reconfigura o modo de voo corrente e continua a execução até que a bateria simulada acabe.

6.4.2 Análise dos resultados

Com respeito às características autonômicas, podemos considerar que o AAP possui as mesmas características já implementadas no Experimento 2. Para que pudéssemos testar as capacidades do sistema de gerar Arquiteturas de Hardware otimizadas usamos diferentes ajustes nos Componentes de Hardware implementados. Uma relação das diferentes opções aos componentes é mostrada na Tabela 6.6.

Como no Experimento 2, a aplicação executará os algoritmos de otimização cada vez que um novo componente é adicionado ao repositório ou dependendo das variáveis de contexto. Os experimentos em relação à geração de arquiteturas foram executados da mesma maneira que no Experimento 2, onde a Figura 6.18 mostra uma sequencia de configurações durante a execução do sistema e a Tabela 6.7 mostra os tempos necessários para geração das Arquiteturas de Hardware geradas durante a execução.

Figura 6.19 - Diagrama de implementação para o Experimento 3.



Fonte: Elaborada pelo Autor.

A tabela mostra que arquiteturas para o tipo de navegação *Heading Hold* são bem mais simples de serem geradas do que arquiteturas para o modo *Waypoint*. Isso é facilmente observável através da observação das complexidades dos algoritmos para ambos os modos de navegação. O tempo de geração para esse caso também inclui o tempo de geração dos IP Cores envolvidos, tempos que podem ser reduzidos quando a geração desses componentes é feita previamente.

Embora o tempo de geração tenha sido no pior caso perto dos vinte minutos, dois terços do tempo de duração da bateria para a simulação feita, os resultados são satisfatórios, pois permitiram a reconfiguração e otimização dos algoritmos de controle da aeronave durante o voo. Especificamente nesta aplicação, as Arquiteturas de Hardware geradas poderiam ter sido pré-armazenadas no sistema antes da decolagem. Porém em aplicações onde o sistema precisasse ser otimizado em função de variáveis, cujos valores não possam ser totalmente previstos, a possibilidade explorada neste experimento poderia ser mais impactante.

A Tabela 6.8 mostra, por fim, o espaço ocupado pelos componentes de observação e comunicação presentes nas Arquiteturas de Hardware geradas neste experimento. Particularmente nesse sistema, a mecânica de geração da interface de comunicação foi importante devido ao fato de que os algoritmos e Arquiteturas de Hardware de navegação não possuem as mesmas interfaces de entrada, embora possuam as mesmas de saída.

Para finalizar a análise do impacto gerado pela implementação da metodologia proposta pelo AHM no Ardupilot foi medido o tamanho do executável final gerado e feita uma análise de desempenho usando medidas de tempo levado para a execução do loop principal de execução do Ardupilot.

- Foi verificado um aumento de 5% no tamanho do executável final após a compilação.
- O loop principal executa, em média, 1% mais lenta, após a adição dos módulos do AHM.

Consideramos que os aumentos em tamanho de executável e redução, relativamente pequena no tempo de execução constituem resultados satisfatórios para esta aplicação.

Tabela 6.6 - Componentes de Hardware, com valores das Funções de Avaliação, implementados para o Experimento 3.

Componente	Resource(c)				Performance(c)
	nRS	*nLS*	*nBRAM*	*nDSP*	
Cordic atan, serial	491	785	0	0	36
Cordic atan- parallel, no pipeline	98	3796	0	0	2
Cordic atan - parallel, maximal	3634	3831	0	0	36
Cordic sincos, serial	550	856	0	0	36
Cordic sincos, parallel - no pipeline	102	3879	0	0	2
Cordic sincos, parallel - maximal	3704	3886	0	0	36
Float adder, low latency	621	563	0	0	8
Float adder, high speed - no usage	552	456	0	0	12
Float adder, high speed - full usage	327	300	0	2	11
Float multiplier, no usage	688	666	0	0	8
Float multiplier, medium usage	369	289	0	1	8
Float multiplier, full usage	179	142	0	2	8
Float multiplier, maximum usage	114	123	0	3	6

Fonte: elaborada pelo Autor.

Tabela 6.7 - Tempo de geração de arquiteturas do sistema, usando variações nos componentes mostrados na Tabela 6.6.

Arquitetura	Tempo de geração
Arquitetura Waypoint A	18:07 min
Arquitetura Waypoint B	19:45 min
Arquitetura Waypoint C	17:12 min
Arquitetura Heading Hold A	9:20 min
Arquitetura Heading Hold B	8:52 min

Fonte: Elaborada pelo Autor.

Através dos resultados, temos que o AAP implementa, também, cinco das sete funcionalidades necessárias a uma aplicação autonômica, podendo, dessa forma ser considerado uma Aplicação Autonômica.

Contando que o AAP, como um sistema autonômico, usando a metodologia de cliente-repositório provida pela arquitetura AHM, podemos concluir que os objetivos secundários OS1 e OS2 foram satisfeitos.

Através da implementação da Arquitetura de Hardware e das análises da execução e tamanho do código compilado, após a adição dos componentes da arquitetura AHM, é possível concluir que o objetivo OS3 foi satisfeito.

Tabela 6.8 - Quantidade de recursos usados pelo hardware de comunicação/observação no Experimento 3.

Arquitetura	Recursos usados	Porcentagem de aumento de área
Arquitetura Waypoint A	340 RLUT	0.1%
Arquitetura Waypoint B	340 RLUT	0.1%
Arquitetura Waypoint C	340 RLUT	0.1%
Arquitetura Heading Hold A	162 RLUT	0.3%
Arquitetura Heading Hold B	162 RLUT	0.3%

Fonte: Elaborada pelo Autor.

Portando, concluímos que o objetivo principal desse experimento foi alcançado, mostrando uma análise do esforço demandado para implementar uma Aplicação Autonômica utilizando o AHM, a partir de um sistema não autonômico preexistente. Reafirmando, assim, as características mostradas nos Experimento 1 e 2. Foi possível ainda realizar uma análise do impacto causado pela implementação da metodologia defendida nesta Tese em um sistema embarcado de referência.

CAPÍTULO 7
CONCLUSÕES

Esse trabalho apresentou uma solução que objetiva possibilitar a manipulação dos componentes de um sistema de hardware autonômico, adaptando-o ao seu contexto de operação, utilizando a técnica de repositório ativo orientado a contexto. Tal objetivo advém da seguinte pergunta de pesquisa:

> ***"Uma vez que as variáveis que influenciam no funcionamento e definem o contexto de um sistema de hardware são mapeadas, como podemos implementar um sistema autonômico que gere novas arquiteturas de hardware como reação a modificações ocorridas em tais variáveis?"***

Para a qual, uma possível solução compõe esta Tese defendida e é definida da seguinte forma:

> **"*Aplicando a técnica de Repositório Ativo Orientado a Contexto, no desenvolvimento de um sistema de hardware implementado em um dispositivo reconfigurável, é possível criar uma arquitetura de hardware/software autonômica, capaz de reagir a alterações no seu contexto de operação, que por sua vez é modelado através de variáveis de contexto*"**

7.1 SUMÁRIO DOS OBJETIVOS ALCANÇADOS

Para nortear o desenvolvimento desta tese, alguns objetivos específicos foram enumerados. A partir dos resultados e implementações apresentados neste documento, consideramos que a Tese atingiu todos os objetivos propostos. Uma arquitetura de repositório ativo e sensível a contexto denominada Autonomic Hardware Manager - AHM foi projetada, implementada e validada. Experimentos

foram planejados, executados e apresentados. Mais detalhadamente, cada objetivo específico é discutido nas próximas seções.

7.1.1 O1 - Escolher um Modelo de Componentes a ser usado para encapsular os componentes de hardware a serem gerenciados.

A escolha do modelo de componentes foi apresentada na seção 5.1 e validada no Experimento 1. Este modelo é baseado naquele já adotado no SystemC, o qual foi expandido. Através de um esquema de descrição em XML, foi possível informar ao repositório quais eram os componentes de hardware presentes no mesmo, bem com sua estrutura interna.

Como demonstrado nos experimentos, o modelo de componentes é capaz descrever um componente de hardware em função de suas entradas, saídas, parâmetros e arquivos dos quais dependa. Usando os resultados enumerados neste objetivo, pudemos comprovar a hipótese secundária "**HS1 - *É possível adaptar um modelo de componentes de software para que possa ser aplicado a componentes de hardware.***".

7.1.2 O2 - Projetar e implementar a arquitetura do Cliente AHM e Repositório AHM.

O projeto e implementação dos componentes do AHM foram bem sucedidos, confirme apresentado na seção de Arquitetura Proposta. As funcionalidades básicas do AHM foram implementadas e validadas inicialmente no Experimento 1, e reafirmada nos Experimentos 2 e 3.

A partir das implementações apresentadas, consideramos esse objetivo O2 satisfeito, comprovando indiretamente as hipóteses secundárias "**HS2 - *O sistema de hardware pode ser reconfigurado, dentro do sistema embarcado, através de interfaces de software.***" e "**HS3 - *É possível gerar novas arquiteturas de hardware para o sistema gerenciado, usando componentes disponíveis, um engenho de software e as ferramentas de síntese de hardware providas pelo fabricante.***".

7.1.3 O3 - Projetar e implementar uma solução que possibilite a observação de elementos internos da arquitetura de hardware.

Esse objetivo engloba o projeto e implementação do Hardware de Comunicação e Observação. As implementações e testes desse componente foram enumeradas durante todos os experimentos realizados uma vez que para implementação de aplicações autonômicas o monitoramento de alguns componentes de hardware é necessário.

Um estudo dos efeitos relativos à adição desses componentes, nos sistemas gerenciados, foi realizado nos Experimento 1 e 2 mostrando que a capacidade de observação dos componentes de hardware não representa grande impacto no tempo de geração ou na quantidade de área ocupada pela arquitetura de hardware.

Portanto consideramos esse objetivo atingido, comprovando a hipótese secundária "**HS4 - *É possível desenvolver uma arquitetura de hardware embarcado em dispositivo reconfigurável na qual seja possível observar o estado de cada componente da arquitetura, independentemente da quantidade de componentes e interfaces presentes na mesma.***".

7.1.4 O4 - Projetar e implementar uma solução de modelagem de contexto para ser implementada nos Cliente AHM e Repositório AHM.

Um metamodelo que permite a modelagem de Variáveis e Eventos relacionados ao Contexto de Operação de aplicações embarcadas foi apresentado no capítulo 5. Testes relacionados à solução de modelagem de contexto foram apresentados nos Experimentos 2 e 3, apresentados no capítulo 6. Nestes experimentos, diferentes aplicações tiveram seus contexos de operação modelados usando a solução proposta.

Com os resultados obtidos, foi possível concluir que o metamodelo escolhido foi suficiente para representar o contexto das aplicações embarcadas, concluindo que o objetivo específico O4 foi alcançado.

7.1.5 O5 - Projetar e implementar uma adaptação dos Cliente AHM e Repositório AHM para incorporar a modelagem de contexto.

Uma vez definido um metamodelo para a representação de Contexto de Operação, foi possível munir a arquitetura do AHM com módulos que possibilitassem a representação do contexto de uma aplicação embarcada. Através desses módulos foi possível descrever variáveis e eventos que descrevem os momentos nos quais a Arquitetura de Hardware gerenciada precisa ser adaptada.

As implementações apresentadas nos Experimentos 2 e 3 comprovam que a Arquitetura AHM, bem como o meta-modelo para representação do Contexto de Operação, é genérica suficiente para ser usada em uma aplicação embarcada orientada à host que possua um canal de comunicação com o Repositório e uma forma de reconfigurar o hardware presente na região reconfigurável.

Ao completar esse objetivo e usando os resultados apresentados ao atingirmos o Objetivo O4, concluímos como verdadeiras as hipóteses secundárias "**HS5 -** ***Uma representação das variáveis de contexto e seus possíveis valores pode ser construída visando fornecer ao sistema um meio de observa-las de forma automática.***" e "**HS6 -** ***Regras de alto nível podem ser escritas através de relações condicionais entre as variáveis que descrevem o contexto e seus valores esperados, visando adaptar o sistema a possíveis condições refletidas em sua descrição de contexto.***".

7.1.6 O6 - Implementação de aplicações de teste para avaliar o desempenho da arquitetura.

Usando a Arquitetura AHM, foram realizados três experimentos: Inicialmente foi mostrado o Hardware Reconfigurable Filter, que foi planejado para

testar as capacidades de geração, descrição e manipulação de hardware providas pela Arquitetura AHM.

O segundo experimento consistiu do projeto e implementação de uma aplicação autonômica para segmentação de imagens, o Autonomic Image Segmentation System. Esse sistema teve como objetivo testar a capacidade da Arquitetura AHM Orientado ao Contexto de possibilitar a implementação de uma aplicação autonômica.

Por último, foi implementado o Autonomic Auto Pilot, um sistema autonômico de gerenciamento do piloto automático de navegação do sistema Ardupilot. Tal experimento visou mensurar o esforço necessário para implementar as funcionalidades de aplicações autonômicas, em uma aplicação não autonômica pré-existente, usando a Arquitetura AHM.

Com as implementações realizadas, esse objetivo pode ser considerado como cumprido, os resultados englobados nesse objetivo foram usados para provas das hipóteses secundárias e para a conclusão dos objetivos anteriores tendo, portanto, impacto indireto em todas as hipóteses que se deseja provar nesta Tese.

7.2 CONTRIBUIÇÕES PRINCIPAIS

Essa Tese atingiu as seguintes contribuições principais:

- Definição de uma Arquitetura que usa o modelo de repositório ativo para implementar sistemas autonômicos.
- Definição de um meta-modelo para descrição de variáveis e eventos relacionados ao Contexto de Operação de uma arquitetura de hardware.
- Definição de uma Arquitetura de gerenciamento que permite a geração automática de Arquiteturas de Hardware através de um Repositório Ativo de Componentes de Hardware.
- Definição de uma Arquitetura para implementação de sistemas autonômicos que possibilita maior flexibilidade e mais funcionalidades autonômicas do que os modelos encontrados previamente na literatura.

7.3 AVALIAÇÃO GERAL

O trabalho proposto nesta Tese envolveu a implementação e validação de uma arquitetura de repositório que projetada, implementada e testada nos experimentos propostos. Devido ao fato de que a área de Sistemas Autonômicos ainda ser bastante recente, não foi possível encontrar, na literatura, outras soluções aplicando o modelo de repositório ativo para sistemas de hardware para que uma comparação direta fosse feita durante a etapa de testes.

No entanto, a Arquitetura proposta possibilita a implementação de forma mais flexível de sistemas autonômicos, se comparada com outros sistemas autonômicos presentes na literatura e apresentados no Capítulo 4.

Dado que os objetivos propostos foram atingidos e hipóteses secundárias comprovadas, concluímos que a hipótese principal é verdadeira e, portanto, também é a Tese proposta e apresentada neste documento.

7.4 TRABALHOS FUTUROS

Os experimentos implementados mostraram avanços no que se diz respeito à usabilidade e funcionalidades da arquitetura AHM, temos como um dos principais trabalhos futuros a divulgação dos resultados apresentados nesta tese a fim de divulgar e consolidar este trabalho.

Outro ponto a ressaltar é que nem todas as capacidades autonômicas foram alcançadas usando a arquitetura AHM. Uma característica que poderia ser mais explorada em futuras publicações é a de auto-reparo. Uma que as situações de falha são identificadas e mapeadas através de Variáveis de Contexto e Eventos, é possível reconfigurar o hardware na região reconfigurável para evitar problemas ou para corrigir problemas em algum Componente de Hardware que tenha apresentado defeito.

Outra característica que poderia ser ainda explorada diz respeito à validação de arquiteturas geradas bem como a funcionalidade de auto-proteção. Como já mencionado nas implementações, ferramentas de teste, já fornecidas pelo fabricante de plataformas de hardware reconfigurável, poderiam ser usadas para

validar as arquiteturas de hardware geradas a fim de avaliar possíveis riscos ou antecipar falhas nas novas arquiteturas.

Um ponto interessante que também não foi abordado nesta Tese, diz respeito à implementação de sistemas autonômicos para indústria automobilística. Poderíamos usar a mesma metodologia do experimento do VANT, adaptando os componentes de controle de acordo com a necessidade da aplicação enquanto encontramos Arquiteturas de Hardware ótimas para as situações de uso das aplicações.

REFERENCIAS BIBLIOGRAFICAS

3DROBOTICS. Mission Planner. **Site do Ardupilot sobre o Mission Planner**. Disponivel em: <http://planner.ardupilot.com/>.

ALTERA CORPORATION. **Fpga run-time reconfiguration: Two approaches**. [S.l.]. 2008.

ALTERA CORPORATION. **Reducing Power Consumption and Increasing Bandwidth on 28-nm FPGAs**. Altera. [S.l.]. 2012.

ALTERA CORPORTAION. Quartus II. **http:** //www.altera.com/products/software/quartus-ii/web-edition/qts-we-index.html.

AN, X. et al. **Autonomic management of reconfigurable embedded systems using discrete control: Application to fpga**. Hal INRIA. [S.l.]. 2013.

ANTONI, L.; LEVEUGLE, R.; FEHÉR, B. **Using run-time reconfiguration for fault injection in hardware prototypes**. Proceedings of17th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2002. DFT 2002. Proceeding. [S.l.]: [s.n.]. 2002. p. 245–253.

AVNET CORPORATION. Zedboard, 2012. Disponivel em: <http://www.zedboard.org>.

BAHETI, R.; GILL, H. Cyber-physical Systems. **The impact of control technology**, p. 161-166, 2011.

BELLOWS, P.; HUTCHINGS, B. **Jhdl-an hdl for reconfigurable systems**. Proceedings on IEEE Symposium on FPGAs for Custom Computing Machines. [S.l.]: [s.n.]. 1998. p. 175 –184.

BIEDERMANN, A. et al. Enhancing fpga robustness via generic monitoring ip cores. **PECCS**, 2011. 379–386.

BLAIR, G.; COUPAYE, T.; STEFANI, J.-B. **Component-based architecture:** the fractal initiative. Annals of Telecommunications Conference. [S.l.]: [s.n.]. 2009. p. 1-4.

BOLCHINI, C. et al. A data-oriented survey of context models. **ACM SIGMOD Records**, 2007. 19-26.

BRITO, A.; SILVEIRA, G.; MELCHER, E. A methodology for modelling and simulation of dynamic and partially reconfigurable systems. **In-Tech Dynamic Modelling**, n. 1, 2010.

BROWN, S.; ROSE, J. Architecture of FPGAs and CPLDs: A Tutorial. In: _____ **IEEE Design and Test of Computers**. [S.l.]: [s.n.], v. 13, 1996. p. 42-57.

C., A. Diy drones-ardupilot. **Diy Drones**, 2010. Disponivel em: <http://diydrones.com/profiles/blogs/ardupilot-main-page>.

COUCH, J. D. **Applications of torc: An open toolkit for reconfigurable computing, Dissertação de Mestrado**. Virginia Polytechnic Institute and State University. [S.l.]. 2011.

COULSON, G. et al. A generic component model for building systems software. **ACM Transactions on Compututing System**, 2008.

DAVID, R. **Two competitive fpga methodologies for run-time reconfiguration**. Altera Corporation. [S.l.]. 2008.

DAYA, B. **Rapid prototyping of embedded systems using field programmable gate arrays, Dissertação de Mestrado**. [S.l.]. 2009.

DELAVAL, G.; MARCHAND, H.; RUTTEN, E. Contracts for modular discrete controller synthesis. **ACM Sigplan Notices**, 2010. 57–66.

DEY, A. K. Understanding and using context. **Personal Ubiquitous Comput**, Janeiro 2001.

DITTRICH, K. R.; GATZIU, S.; GEPPERT, A. The active database management system manifesto: A rulebase of adbms features. **Rules in Database Systems**, 1995. 1-17.

EL-ARABY, E.; GONZALEZ, I.; EL-GHAZAWI, T. Exploiting partial runtime reconfiguration for high-performance reconfigurable computing. **ACM Trans. Reconfigurable Technol**, 2009.

ETO, E. **Difference-Based Partial Reconfiguration**. Xilinx Corporation. [S.l.]. 2007.

GNU. GNU Make. Disponivel em: <http://www.gnu.org/software/make/>.

GROTKER, T. **System design with systemc**. Kluwer Academic Publishers. [S.l.]. 2002.

GUCCIONE, S. A.; LEVI, D. **Jbits: A java-based interface to fpga hardware**. Xilinx. [S.l.]. 1998a.

GUCCIONE, S. A.; LEVI, D. **Xbi:** A java-based interface to fpga hardware. Procedings of Configurable Computing Technology and its use in High Performance Computing, DSP and Systems Engineering. [S.l.]: [s.n.]. 1998b.

GUILLET, S. et al. **Designing formal reconfiguration control using uml/marte**. 7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC). [S.l.]: [s.n.]. 2012. p. 1-8.

GUILLET, S. et al. **Modeling and synthesis of a dynamic and partial reconfiguration controller**. 22nd International Conference on Field Programmable Logic and Applications (FPL). [S.l.]: [s.n.]. 2012. p. 703-706.

HAYKIN, S. **Adaptive Filter Theory**. 3ª Edição. ed. [S.l.]: Prentice-Hall, 1996.

HAYKIN, S. **Neural Networks:** A Comprehensive Foundation. 2ª Edição. ed. [S.l.]: Prentice Hall, 1998.

HOFFMANN, H. et al. **Using code perforation to improve performance, reduce energy consumption, and respond to failures, Relatório Técnico**. MIT. [S.l.]. 2009.

HOFFMANN, H. et al. **Application heartbeats:** a generic interface for specifying program performance and goals in autonomous computing environments. Proceedings of the 7th international conference on Autonomic computing. [S.l.]: [s.n.]. 2010. p. 79–88.

HORN, P. **Autonomic computing: Ibm's perspective on the state of information technology**. [S.l.]. 2001.

HORTA, E.; LOCKWOOD, J. W. **Parbit: a tool to transform bitfiles to implement partial reconfiguration of field programmable gate arrays**. Washington University. [S.l.]. 2001.

HUIYUAN, Z.; JUN, Z.; ZHE, H. **Architecture design of traffic monitoring system**. International Conference on ITS. [S.l.]: [s.n.]. 2007. p. 1 - 4.

IBM. **An architectural blueprint for autonomic computing, IBM White Paper**. [S.l.]. 2003.

IEEE. **IEEE robotic trends 2012**. IEEE. [S.l.]. 2012.

INSAURRALDE, C. C. **utonomic computing management for unmanned aerial vehicles**. Digital Avionics Systems Conference (DASC). [S.l.]: [s.n.]. 2013.

JOVANOVIC´, S.; TANOUGAST, C.; WEBER, S. A new self-managing hardware design approach for fpga-based reconfigurable systems. **Reconfigurable Computing: Architectures, Tools and Applications**, 2008. 160–171.

KAO, C. Benefits of Partial Reconfiguration. **Xcell Journal**, 2005. 65-67.

KELLER, E. **Jroute:** A run-time routing api for fpga hardware. Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing. [S.l.]: [s.n.]. 2000. p. 874– 881.

KLABUNDE, C. **Um Estudo sobre Componentes de Software, Relatório Técnico**. Universidade Federal do Rio Grande do Sul. [S.l.]. 1999.

KOESTER, M. et al. Design optimizations for tiled partially reconfigurable systems. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, p. 1048 – 1061, 2011.

KUEHNLE, M. et al. **The study of a dynamic reconfiguration manager for systems-on-chip**. IEEE Computer Society Annual Symposium on VLSI. [S.l.]: [s.n.]. 2011. p. 13 –18.

L.MCGUINNESS, D.; HARMELEN, F. V. **Owl web ontology language overview, W3C Recomendation**. World Wide Web Consortium (W3C). [S.l.]. 2004.

LAMINAR RESEARCH. **Site do XPlane-10**. Disponivel em: <http://www.x-plane.com/desktop/home/>.

LANUSSE, A. et al. **Papyrus uml:** an open source toolset for mda. Proc. of the Fifth European Conference on Model-Driven Architecture Foundations and Applications. [S.l.]: [s.n.]. 2009. p. 1–4.

LAU, K.-K.; WANG, Z. Software Component Models. **IEEE Transactions on Software Engineering**, 2007. 709-724.

LEE et al. Hardware context-switch methodology for dynamically partially reconfigurable systems. **Journal of Information Science and Engineering**, p. 1289–1305, 2010.

LEE, E. A. **Cyber physical systems:** Design challenges. IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC), 2008. [S.l.]: [s.n.]. 2008. p. 363-369.

LING, X. P.; AMANO, H. **WASMII:** a data driven computer on a virtual hardware. Proceedings of IEEE workshop on FPGAs for custom computing machines. [S.l.]: [s.n.]. 1993.

LUCRÉCIO, D.; PRADO, A.; SANTANA, E. **A Survey on Software Components**. Procedings of IEEE Eromicro. [S.l.]: [s.n.]. 2004.

MCMILLAN, S.; GUCCIONE, S. A. **Partial run-time reconfiguration using jrtr**. Proceedings of the The Roadmap to Reconfigurable Computing. [S.l.]: [s.n.]. 2000. p. 352-360.

MERMOUD, G. **A Module-Based Dynamic Partial Reconfiguration Tutorial, Technical report**. Ecole Polytechnique Federale de Lausanne. [S.l.]. 2004.

MEROBASE CORPORATION. Merobase - software component finder. Disponivel em: <www.merobase.com>.

NAMI, M. R.; BERTELS, K. **A Survey of Autonomic Computing Systems**. Third International Conference on Autonomic and Autonomous Systems. [S.l.]: [s.n.]. 2007. p. 19-25.

NAMI, M. R.; SHARIFI, M. A survey of autonomic computing systems. **Intelligent Information Processing III**, 2007. 101-110.

OBJECT MANAGEMENT GROUP INC. **A uml profile for marte (version beta 1)**. Object Management Group. [S.l.]. 2007.

PAPADIMITRIOU, K.; DOLLAS, A. A. A. An effective framework to evaluate dynamic partial reconfiguration in fpga systems. **IEEE Transactions on Instrumentation and Measurement**, p. 1642 –1651, 2010.

PARK, S. R.; BURLESON, W. **Reconfiguration for power saving in real-time motion estimation**. Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing. [S.l.]: [s.n.]. 1998. p. 3037–3040.

PEREIRA, R. **Suporte ao desenvolvimento e uso de frameworks e componentes, Tese de Doutorado**. Universidade Federal do Rio Grande do Sul. [S.l.]. 2000.

POETTER, A. et al. Jhdlbits: The merging of two worlds. **Field Programmable Logic and Application**, p. 414–423, 2004.

PUTTEGOWDA, K. **Context switching strategies in a run-time reconfigurable system, Dissertação de mestrado**. Virginia Polytechnic Institute and State University. [S.l.]. 2002.

RESANO, J. et al. Efficiently scheduling runtime reconfigurations. **ACM Transaction on Automotive Electronic Systems**, 2008. 1–58.

ROSINGER, H.-P. **Connecting customized ip to the microblaze soft processor using the fast simplex link (fsl) channel**. Xilinx. [S.l.]. 2004.

SANTAMBROGIO, M. D. et al. **Enabling technologies for self-aware adaptive systems**. Conference on Adaptive Hardware and Systems (AHS). [S.l.]: [s.n.]. 2010. p. 149–156.

SCALERA, S. M. **Context switching reconfigurable computing, Relatório Técnico**. DTIC. [S.l.]. 2001.

SCHILIT, B.; ADAMS, N.; WANT, R. **Context-Aware Computing Applications**. Proceedings of the 1994 First Workshop on Mobile Computing Systems and Applications. [S.l.]: [s.n.]. 1994.

SCHILIT, B.; ADAMS, N.; WANT, R. **Context-Aware Computing Applications**. roceedings of the 1994 First Workshop on Mobile Computing Systems and Applications. [S.l.]: [s.n.]. 1995. p. 85-90.

SCILAB. wfir. **Site da Scilab Enterprises**. Disponivel em: <https://help.scilab.org/docs/5.5.2/en_US/wfir.html>.

SEDCOLE, P. et al. **Modular dynamic reconfiguration in Virtex FP-GAs**. IEE Proceedings Computers and Digital Techniques. [S.l.]: [s.n.]. 2006. p. 157-164.

SIDIROPOULOS, H. et al. **On supporting efficient partial reconfiguration with just-in-time compilation**. 19th Reconfigurable Architectures Workshop. [S.l.]: [s.n.]. 2012.

SIRONI, F. et al. **Self-aware adaptation in fpga-based systems**. International Conference on Field Programmable Logic and Applications (FPL). [S.l.]: [s.n.]. 2010.

SOHANGHPURWALA, A. A. **Openpr: An open-source partial reconfiguration toolkit for xilinx fpgas, Dissertação de mestrado**. Virginia Polytechnic Institute and State University. Blacksburg. 2010.

STANISLAV, T.; MICLEA, L. Cyber-physical systems-concept, challenges and research areas. **Journal of Control Engineering and Applied Informatics**, 2012. 28-33.

SZYPERSKI, C. **Component Software – Beyond Object-Oriented Programming**. [S.l.]: Addison-Wesley Longman Publishing Co., v. 2, 2002.

TRIMBERGER, S. et al. **A Time-Multiplexed FPGA**. Proceedings of IEEE symposium on FPGAs for custom computing machines. [S.l.]: [s.n.]. 1997.

XILINX CORPORATION. **Logicore ip cordic v5.0**. Xilinx. [S.l.]. 2010.

XILINX CORPORATION. Xilinx University Program XUPV5-LX110T Development System, 2011. Disponivel em: <http://www.xilinx.com/univ/xupv5-lx110t.htm>.

XILINX CORPORATION. **Lowering Power at 28 nm with Xilinx 7 Series Devices**. Xilinx. [S.l.]. 2015.

XILINX CORPORATION. PlanAhead. Disponivel em: <http://www.xilinx.com/tools/planahead.html>.

YAN, N.; ZHANG, Y.; LALA, L. **Road monitoring and traffic control system design**. International Conference on Information Engineering and Computer Science. [S.l.]: [s.n.]. 2009.

YE, Y. **Supporting component-based software development with active component repository systems, Tese de Doutorado**. University of Colorado. Boulder, CO, USA. 2001.

APÊNDICE A - IMPLEMENTADO A SOLUÇÃO AHM EM UM SISTEMA DE HARDWARE ORIENTADO À PLATAFORMA

A intenção desta seção é mostrar como o AHM pode ser adicionado e usado em um sistema de hardware orientado à plataforma. Para tanto, usaremos as ferramentas e modelos providos pela Xilinx, uma vez que todas as implementações desta Tese foram feitas usando essas ferramentas.

Seja qual for o fabricante, é possível utilizar a solução proposta pelo AHM se Plataforma de Hardware possuir as seguintes características:

- Capaz de executar software e que possa se comunicar com o hardware configurado na região reconfigurável.
- Capaz de armazenar os *bitstreams* das Arquiteturas de Hardware recebidas.
- Forneça um meio de comunicação com o Repositório AHM.

Além disso a Plataforma do Repositório, onde o Repositório AHM executa e outras aplicações que utilizem o mesmo, precisa prover os seguintes requisitos:

- A plataforma precisa de um meio de acessar as Ferramentas de Síntese do fabricante através de APIs de software, linguagem de script ou executáveis externos.
- O sistema repositório precisa fornecer um meio de comunicação compatível com sistema Cliente.

Uma vez garantidos os requisitos dos sistemas cliente e repositório, o código relativo ao AHM Cliente e Repositório podem ser encontrados no *Github* (https://github.com/juliocpmelo/AHM). Com os códigos relativos ao AHM, uma aplicação usando as funcionalidades providas nesta Tese pode ser implementada se os seguintes passos forem seguidos para a criação do sistema cliente:

1. Implementar o sistema de hardware e software usando as ferramentas fornecidas pelo fabricante.
2. Projetar uma ou mais regiões reconfiguráveis que serão gerenciadas pelo AHM, descrever as regiões planejadas usado o formato XML especificado pelo AHM. Resultando em um arquivo como mostra a Figure A.1.
3. Projetar, opcionalmente, um hardware que irá executar, inicialmente, na região reconfigurável. Esse hardware é referenciando através da tag

defaultArchitecture como mostrado na Figura A.1 e descrito seguindo o modelo XML proposto pelo AHM.

4. Usar as ferramentas de geração de Hardware de Comunicação e Observação, para criação do componente de Hardware de Comunicação que deve ser adicionado no projeto. Para o exemplo, os componentes gerados são exibidos na Figura A.1.
5. Criar um componente de barramento, denominado, <nome_da_regiao_reconfiguravel>_buswrappe com as mesmas saídas externas projetadas para a região reconfigurável, para acessar, no mínimo as quatro interfaces principais do hardware de comunicação. Adicionar o componente ao sistema de hardware projetado, como exibido na Figura A.3.
6. Adicionar à descrição da região reconfigurável o endereço base, na memória do sistema embarcado, através da tag buswrapperBaseAddr.
7. Realizar as conexões dos componentes gerados no passo 5 aos seus respectivos buswrappers no passo 6, fazendo as conexões das interfaces externas e das cinco interfaces necessários ao hardware de comunicação seguindo a ordem: commHardware_address, commHardware_dataIn, commHardware_dataOut, commHardware_mode e commHardware_clk.
8. Sintetizar o hardware para gravação no dispositivo reconfigurável.

Figura A. 1 - Arquivo APController_Zed descrevendo as regioes reconfiguraveis contidas no projeto.

```
<hardwareProject name="APControl_Zed" path="/home/jullo/Documents/APController_Zed">

  <reconfigurableRegion name="APNavigationControl" buswrapperBaseAddr="none">
    <communicationHardware>
    </communicationHardware>
    <!--
      this reconfigurable region has no external inputs/outputs except the default ones
    -->
    <defaultArchitecture file="/home/jullo/Documents/APController_Zed/APNavigationControlConfig1.xml"/>
  </reconfigurableRegion>

  <reconfigurableRegion name="PayloadProcessor" buswrapperBaseAddr="none">
    <communicationHardware>
      <output name="ap_idle" type="bit"/>
    </communicationHardware>
    <defaultArchitecture file="/home/jullo/Documents/APController_Zed/PayloadProcessorBB.xml"/>
  </reconfigurableRegion>
</hardwareProject>
```

Fonte: Elaborada pelo Autor.

Figura A. 2 - a) Componente gerado para a região reconfigurável APNavigationControl; b) Componente gerado para a região reconfigurável PayloadProcessor.

```
-- This file was auto generated by the CommunicationHardwareGenerator

-- Component Declaration

component APNavigationControl_communicationHardware is
generic (
    commHardware_addressWidth : integer := 32;
    commHardware_dataWidth : integer := 32
);
port (
    commHardware_address: in std_logic_vector(commHardware_addressWidth - 1 downto 0);
    commHardware_clk: in std_logic;
    commHardware_dataIn: in std_logic_vector(commHardware_dataWidth - 1 downto 0);
    commHardware_dataOut: out std_logic_vector(commHardware_dataWidth - 1 downto 0);
    commHardware_mode: in std_logic
);
end APNavigationControl_communicationHardware;

-- Component Declaration end
```

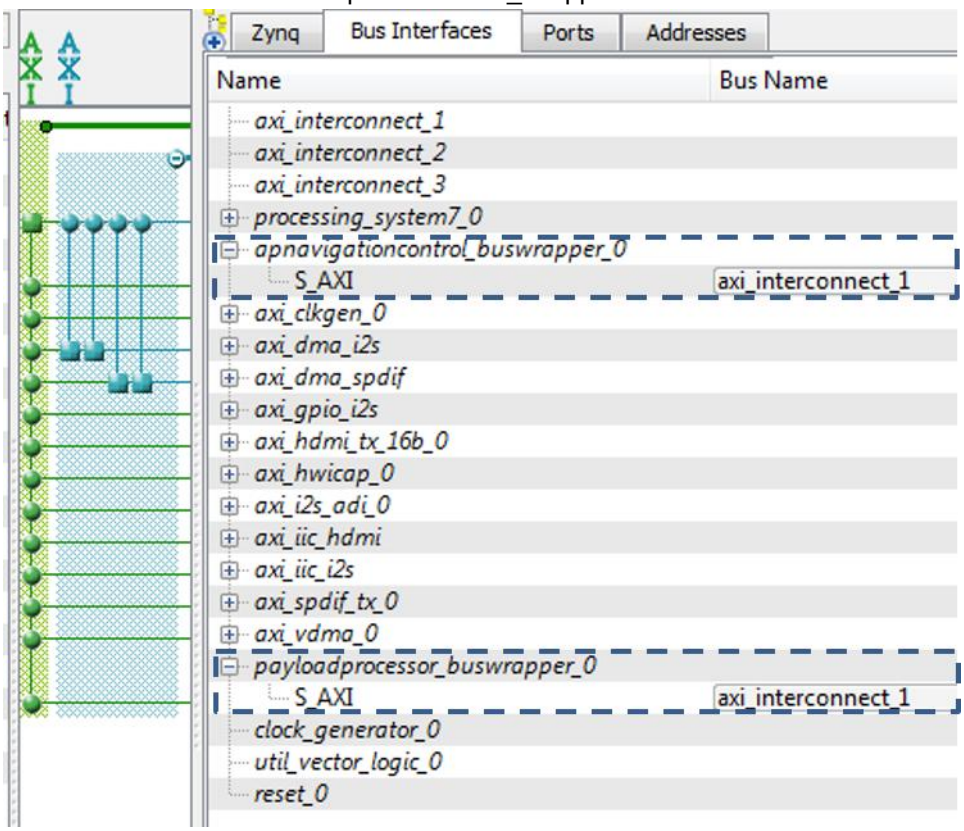
a)

```
-- This file was auto generated by the CommunicationHardwareGenerator

-- Component Declaration

component PayloadProcessor_communicationHardware is
generic (
    commHardware_addressWidth : integer := 32;
    commHardware_dataWidth : integer := 32
);
port (
    -- external inputs/outputs
    ap_idle : out std_logic

    --default inputs/outputs
    commHardware_address: in std_logic_vector(commHardware_addressWidth - 1 downto 0);
    commHardware_clk: in std_logic;
    commHardware_dataIn: in std_logic_vector(commHardware_dataWidth - 1 downto 0);
    commHardware_dataOut: out std_logic_vector(commHardware_dataWidth - 1 downto 0);
    commHardware_mode: in std_logic
);
end PayloadProcessor_communicationHardware;

-- Component Declaration end
```

b)

Fonte: Elaborada pelo Autor.

Figura A. 3 - Adicionando os componentes bus_wrapper usando a ferramenta Xilinx XPS.



Fonte: Retirada do programa XPS (XILINX CORPORATION)

Para a implementação da aplicação no Repositório, as bibliotecas disponíveis no github, previamente mencionadas, devem ser incluídas por uma aplicação que execute em uma Plataforma que disponha das ferramentas do Fabricante Xilinx. Uma vez incluídas no código da Aplicação no Repositório, as APIs precisam dos seguintes elementos básicos para funcionar corretamente:

- Descrição em XML de alguma Base de Dados de Componentes, seguindo o modelo do AHM como, por exemplo, mostra a Figura A. 4.
- Descrição em XML das variáveis e eventos de contexto em uma Base de Dados de Informação de Contexto, como, por exemplo, mostrado nos XML D.1 e XML D.2.

Com esses elementos a Aplicação do Repositório pode ser executada e, em conjunto com a Aplicação Embarcada, irá realizar as atividades de monitoramento e sensibilidade ao contexto projetadas para o AHM.

Figura A. 4 - Exemplo de Base de Dados de Componentes para o AAP

```
<!--indicates a set of components inside the same document-->
<componentBase>

	<!--Component tag defines the component name and its main file-->
	<component name="APWP_Controller" file="/home/julio/ArdupilotAccel/
		L1_Controll/APWP_Controller.vhdl">


	<!--Dependency tag defines the files needed to be in the same
directory of the given component in order to compile
	 optionally the type flag could specify if a file is an ipcore,
thus the software will add it to the complition differently from an vhd
file-->
	<dependency file="/home/julio/ArdupilotAccel/
	 L1_Controll/APWP_Controller_sm.vhd"/>
	<dependency file="/home/julio/ArdupilotAccel/
	L1_Controll/operation_mux.vhd"/>
	<dependency file="/home/julio/ArdupilotAccel/
	L1_Controll/fixed_adder.vhd"/>
	<dependency file="/home/julio/ArdupilotAccel/
	L1_Controll/fixed_point_comparator.vhd"/>
	<dependency file="/home/julio/ArdupilotAccel/
	L1_Controll/Vector2DOperations.vhd"/>
	<dependency file="/home/julio/ArdupilotAccel/
	L1_Controll/results_storage.vhd"/>
	<dependency file="/home/julio/ArdupilotAccel/
	L1_Controll/TypeDeclarations.vhd"/>
	<dependency file="/home/julio/ArdupilotAccel/
	L1_Controll/Vector2DOperationsControl_sm.vhd"/>
	<dependency file="/home/julio/ArdupilotAccel/
	L1_Controll/VectorOp_types.vhd"/>

	<dependency type="xilinx-ipcore"
	file="/home/julio/ArdupilotAccel/
	L1_Controll/ipcore_dir/cordic.ngc"/>
	<dependency type="xilinx-ipcore"
	file="/home/julio/ArdupilotAccel/
	L1_Controll/ipcore_dir/cordic_sincos.ngc"/>
	<dependency type="xilinx-ipcore"
	file="/home/julio/ArdupilotAccel/
	L1_Controll/ipcore_dir/fixed_to_float.ngc"/>
	<dependency type="xilinx-ipcore"
	file="/home/julio/ArdupilotAccel/
	L1_Controll/ipcore_dir/Float_add.ngc"/>
	<dependency type="xilinx-ipcore"
	file="/home/julio/ArdupilotAccel/
	L1_Controll/ipcore_dir/float_compare.ngc"/>
	<dependency type="xilinx-ipcore"
	file="/home/julio/ArdupilotAccel/
	L1_Controll/ipcore_dir/float_divider.ngc"/>
	<dependency type="xilinx-ipcore"
	file="/home/julio/ArdupilotAccel/
	L1_Controll/ipcore_dir/Float_mult.ngc"/>
	<dependency type="xilinx-ipcore"
	file="/home/julio/ArdupilotAccel/
	L1_Controll/ipcore_dir/float_sqrt.ngc"/>
	<dependency type="xilinx-ipcore"
	file="/home/julio/ArdupilotAccel/
```

```
        L1_Controll/ipcore_dir/Float_to_Fixed.ngc"/>

        <!--Component input and outputs-->
        <!--Component input bit type (AKA std_logic for vhdl)-->
        <input name="clk" type="bit"/>
        <!--Component input vector type (AKA std_vector in vhdl)-->
        <input name="gsVectorX" type="vector" startIndex="31"
         op="downto" endIndex="0"/>
        <input name="gsVectorY" type="vector" startIndex="31"
         op="downto" endIndex="0"/>
        <input name="airUnitVectorX" type="vector" startIndex="31"
         op="downto" endIndex="0"/>
        <input name="airUnitVectorY" type="vector" startIndex="31"
         op="downto" endIndex="0"/>
        <input name="ABVectorX" type="vector" startIndex="31"
         op="downto" endIndex="0"/>
        <input name="ABVectorY" type="vector" startIndex="31"
         op="downto" endIndex="0"/>
        <input name="groundspeed" type="vector" startIndex="31"
         op="downto" endIndex="0"/>
        <input name="K_L1" type="vector" startIndex="31"
         op="downto" endIndex="0"/>
        <input name="L1_dist" type="vector" startIndex="31"
         op="downto" endIndex="0"/>
        <input name="L1_dist_min" type="vector" startIndex="31"
         op="downto" endIndex="0"/>
        <input name="sin_nu1_max" type="vector" startIndex="31"
         op="downto" endIndex="0"/>
        <input name="sin_nu1_min" type="vector" startIndex="31"
         op="downto" endIndex="0"/>
        <input name="nu_limit" type="vector" startIndex="31"
         op="downto" endIndex="0"/>
        <input name="max_nu" type="vector" startIndex="31"
         op="downto" endIndex="0"/>
        <input name="min_nu" type="vector" startIndex="31"
         op="downto" endIndex="0"/>
        <input name="nav_bearing_cd_condition" type="bit"/>
        <input name="wpa_reference" type="bit"/>

        <output name="nu" type="vector" startIndex="31"
          op="downto" endIndex="0"/>
        <output name="last_nu" type="vector" startIndex="31"
          op="downto" endIndex="0"/>
        <output name="lateral_acceleration_dem"
          type="vector" startIndex="31" op="downto" endIndex="0"/>
        <output name="bearing_error" type="vector" startIndex="31"
          op="downto" endIndex="0"/>
        <output name="nav_bearing" type="vector" startIndex="31"
          op="downto" endIndex="0"/>
        </component>

</componentBase>
```

Fonte: Elaborada pelo Autor.

APÊNDICE B - ALGORITMO USADO PARA GERAÇÃO DOS ARQUIVOS QUE DESCREVEM OS COMPONENTES DE HARDWARE

Codigo B.1: Pseudo código mostrando algoritmo de criação automática das Arquiteturas de Hardware.

```
-- generates the child HDLs of a given "topComponent"
-- returning a vector containing all files related to
-- that component.
function generateChildHDL(topComponent)
      -- the file of the current component
      componentFile = file(topComponent.instanceName())
      -- a vector of hdl files related to this component
      hdlFiles.insert(componentFile)

      for comp in topComponent.childComponents do
       -- if this child is not static specified
       -- through the component database, its files
       -- need to be generated as well
       if(!comp.isDataBaseComponent()) then
       hdlFiles.insert(generateChildHDL(comp))
       end
      end


      -- loop over the children of the current component
      -- generate the related instances and make the
      -- required connections
      for comp in topComponent.childComponents() do
       componentFile.addInstance(topComponent)
       for port in comp.getPorts() do
       if port.isConnected(topComponent) then
       connecetPort = comp.
                                        getConnectedPort(topComponent)
       componentFile.insertConnection(topComponent,
                                comp,connecetPort)
       else if(port.isSwInterface()) then
       topComponent.addIntefarce(port)
       topComponent.setSwInterface(port)
       else
       comp2 = port.getConnectedComponent()
       if (comp2 != NULL) then
       componentFile.insertConnection(comp,comp2,
            connectedPort)
       end
       end
       end
      end

      return hdlFiles
end function

-- generates the HDLFile to the given the given
```

```
-- "topComponent". Returns a vector containing all
-- files related to the generated HDL.
function generateMainHDL(commComponent, topComponent)
      vector<file> dependencies;

      -- main hdl file
      mainHdlFile = file(commComponent.name + ".vhd")

      for comp in topComponent.childComponents do
       dependencies.insert(comp.getDependencies())
      end

      -- vector containing all hdl files related to this
      -- topComponent
      childrenHDL = generateChildHDL(topComponent)

      dependencies.insert(childrenHDL)
      dependencies.insert(mainHdlFile)

      -- vector containing a list of Port structure
      ports = topComponent.getPorts()

      -- vector containg the softwareAccessible input
      -- ports
      vector<Port> swInPorts

      -- vector containg the softwareAccessible output
      -- ports
      vector<Port> swOutPorts

      -- generates a instance of this component in the
      -- main hdl file
      mainHdlFile.addInstance(topComponent)

      -- loop over all ports and find which are connected
      -- to external interfaces through the communication
      -- hardware and which are software accessible ones
      for topComponentPort in ports do
       if topComponentPort.isConnected(commComponent) then
       connecetPort = commComopnent.
                                 getConnectedPort(topComponentPort)
       hdlFile.insertConnection(topComponentPort,
                                              connecetPort)
       else if topComponentPort.isSwInteface() then
       if(topComponentPort.type = "in") then
       swInPorts.insert(topComponentPort)
       else if(topComponentPort.type = "out") then
       swOutPorts.insert(topComponentPort)
       end
       mainHdlFile.
       insertRegisterConnection(topComponentPort)
       mainHdlFile.addRegister(topComponentPort)
       end
      end

      -- create an hdl hardware process and add the
      -- swInterfaces which are "in" type in this process
```

```
      mainHdlFile.createWriteProcess()
      addr = 0
      for port in swInPorts do
       mainHdlFile.addPortToWriteProcess(addr,port)
       addr = addr + 1
      end

      -- create an hdl hardware process and add the
      -- swInterfaces which are "out" type in this process
      mainHdlFile.createReadProcess()
      for port in swOutPorts do
       mainHdlFile.addPortToReadProcess(addr,port)
       addr = addr + 1
      end

      return dependencies
end

-- returns generates all files related to the hardware
-- of a specific Architecture
function generateConfigurationFiles(Architecture arch)
      -- ReconfigurableRegion structure
      region = arch.getAssignedReconfigurableRegion()
      -- HardwareComponent structure
      commComponent = region.getCommunicationHardware()

      -- vector containing all hdl files related to this
      -- architecture
      files = generateMainHDL(commComponent, arch.topComponent)

      -- add the depenendcy files to the current hardware
      -- project in order to perform the file generation
      hardwareProject.addDependencyFiles(files)

      -- static call to the HardwareGenerationModule class
      HardwareGenerationModule.
       sythesizer.
       synthesize(hardwareProject,arch,
            hardwareProject.deviceDescription)

      -- static call to the HardwareGenerationModule class
      HardwareGenerationModule.
       placerAndRouter.
       route(hardwareProject,arch,
       hardwareProject.deviceDescription,
       architecture.timingConstraints)

      -- static call to the HardwareGenerationModule class
      HardwareGenerationModule.
       bitstreamGenerator.
       generateBitstream(hardwareProject,arch)
end
```

APÊNDICE C - ALGORITMO QUE GERA AS DESCRIÇÕES DOS HARDWARES DE OBSERVAÇÃO

Codigo C.1: Pseudo código mostrando algoritmo de criação automática dos Hardware de Observação.

```
-- create an hardwareObserver for the specific context
-- variable and add the hardware to the given  Architecture.
function createObserver(contextVar, observerPort,
                        arch)
  -- reference to the architectures top component
  topComponent = arch.topComponent

  -- vector with the components to process
  componentsToProcess

      -- observercomponent to be generated
  observedComponent

      -- port related to the observer component
  sc_port observedPort

      -- creates a hardware component with the parameters
      -- required according the type of observer
      observer = createObserverComponent(contextVar.observerId,
            contextVar.observerType)

  if(contextVar.observerType = "observer") then
      -- if it is an "observer" type it is referencing
      -- already added observers thus we don't need
      -- any processing
      connectObserver(topComponent, NULL, observer, contextVar)
  else
      -- otherwise we need to add wrappers if needed
      -- between the observed component and the topComponent
      componentsToProcess.
       insert(topComponent.getChildHardwareComponents())
      while !componentsToProcess.empty() do
            HardwareComponent comp = componentsToProcess.front()
            if comp.instanceName = contextVar.componentName then
                  observedComponent = comp
       observedPort = comp.getInterface(
                                    contextVar.componentPort)
                  componentsToProcess.clear()
                  break
            end
            componentsToProcess.insert(comp.getChildHardwareComponents())
            componentsToProcess.remove_front()
      end


      comp = observedComponent.getParentComponent()

      -- when the parent is not the top component we must
      -- go upwards the architecture wrapping the observed
      -- output/input until we reach the topComponent which
      -- will be the access point for all observers
```

```
      while comp != topComponent do
            HardwareComponent parent = comp.getParentComponent()
       addObserverPort(parent, observedPort, contextVar)
       portConnect(observedPort,
                        parent.getInterface("observer_" + ontextVar.
       observerName))
       observedPort = parent.getInterface("observer_" + contextVar.
                  observerName)
       comp = comp.getParentComponent()
      end
      connectObserver(topComponent, observedPort,
           observer, contextVar)
  end
  topComponent.addChildComponent(observer)
end

-- adds a port in the specified component reffering to
-- the specified context variable
function addObserverPort(comp, wrappedPort, contextVar)
      -- port structure
      if portWrapper != NULL then
       p = createPort("observer_" + contextVar.observerName,
          wrappedPort.type,"out")
      else
       p = createPort("observer_" + contextVar.observerName,
          "bit_type","out")
       p.setSwInterface("true")
      end
      p.addAttribute("observer",contextVar)
      p.addAttribute("isObserverPort", "true")
end

-- connects a given observer to the specified
-- topComponent.
function connectObserver(HardwareComponent topComponent,
        sc_port observedPort,
        HardwareComponent observer,
        ContextVariable contextVar)
      -- for a observer of "observer" type we must search the
      -- two referenced observers first an then connect their
      -- outputs to the inputs of the new observer. For this
      -- case, the Observers referenced by "observer1" and
      -- "observer2" values must have been already added.
      if(contextVar.observerType = "observer") then
         -- Port objects
        observer1Port, observer2Port
        for p in topComponent.getAllPorts() do
            if p.observer.name == contextVar.observer1 and
               p.isObserverPort() then
              observer1Port = p.getConnectedInterface()
            else if(p.isObserverPort() and
                    p.observer.name == contextVar.observer2) then
              observer2Port = p.getConnectedInterface()
            end
        end
        portConnect(observer1Port, observer.getInterface("observer1"))
        portConnect(observer2Port, observer.getInterface("observer2"))
    -- for a Observer of "value" type just the connection
    -- needs to be done between port and Observer
```

```
    else if contextVar.observerType = "value" then
      portConnect(observedPort,
                  observer.getInterface("observerInterface1"))
      -- for a Observer of "interface" type we need to search
      -- for the other observer interface to be compared then
      -- make the connection. For this case, the Observer
      -- referenced by "observerInterface2" value must
      -- have been already added.
    else if contextVar.observerType = "interface" then
      for p in topComponent.getAllPorts() do
        if(p.isObserverPort() and
          p.observer.name == contextVar.interfaceName) then
          observerInterface2 = p.getConnectedInterface()
          portConnect(observedPort,
                      observer.getInterface("observerInterface1"))
          portConnect(observerInterface2,
           observer.getInterface("observerInterface2"))
          break
        end
      end
    end
    addObserverPort(topComponent, NULL, contextVar)
    portConnect(observer.getInterface("observerOutput"),
           topComponent.getPort("observer_" +
                                    contextVar.observerName))
end

-- generates the observer hardwares for each context
-- variable that requires it. Returns a vector containing
-- all files generated.
function generateObserverHardware(arch, contextVariables )

      for v in variables do
       if v.isHardwareType then
       observerComponent = createObserver(contextVar,
                                                  observerPort, arch)
       file = file(observerComponent.getName + ".vhd")
       generatedHdlFiles.insert(file)
       end
      end

      return generatedHdlFiles
end
```

APÊNDICE D - DESCRICAO DE CONTEXTO DE OPERACAO, EVENTOS DE CONTEXTO E CALLBACK DA APLICACAO AISS

XML D.1: Definição das variáveis que representam o contexto de operação para aplicação AISS.

```
<operatingContext name="ASSContextVariables">
      <contextVariable name="FilterType" type="Application"/>
      <contextVariable name="LighteningConditions" type="Application"/>
      <contextVariable name="NumberOfClusters" type="Application"/>
      <contextVariable name="ThresholdValues" type="Application"/>
      <contextVariable name="BatteryLevel" type="Application"/>
      <contextVariable name="EnergySupply" type="Application"/>
      <contextVariable name="PowerProfile" type="Compound">
       <simpleCondition contextVariable="EnergySupply"
                              comparator="eq" value="true">
       <conditionResult value="HIGH"/>
       </simpleCondition >
       <compoundCondition logicOperation="and">
       <simpleCondition contextVariable="EnergySupply"
                                    comparator="eq" value="false"/>
       <compoundCondition logicOperation="and">
       <simpleCondition contextVariable="BatteryLevel"
                                          comparator="gt" value="130"/>
       <simpleCondition contextVariable="BatteryLevel"
                                          comparator="lt" value="255"/>
       </compoundCondition >
       <conditionResult value="MEDIUM"/>
       </compoundCondition >

       <compoundCondition logicOperation ="and">
       <simpleCondition contextVariable ="EnergySupply"
                                    comparator="eq" value="false"/>
       <simpleCondition contextVariable ="BatteryLevel"
                                    comparator="lt" value="130"/>
       <conditionResult value="LOW"/>
       </compoundCondition >
      </contextVariable >
</operatingContext >
```

XML D.2: Definição dos eventos de Contexto da aplicação AISS.

```
<contextEvent id="lighteningChangeThreshold1">
      <compoundCondition logicOperation="and">
       <simpleCondition contextVariable="LighteningConditions"
                              comparator ="gte"value="0.3"/>
       <simpleCondition contextVariable="LighteningConditions"
                              comparator="lte"value="0.6"/>
       <simpleCondition contextVariable="NumberOfClusters"
                              comparator="lt"value="4"/>
       <action actionId="lighteningChangeThreshold_A1"
                     callback="recomputeThreshold"/>
      </compoundCondition>
</contextEvent>

<contextEvent id="lighteningChangeThreshold2">
```

```
      <compoundCondition logicOperation="and">
       <simpleCondition contextVariable="LighteningConditions"
                        comparator="lt"value="0.3"/>
       <simpleCondition contextVariable="NumberOfClusters"
                        comparator ="lt"value="4"/>
       <action actionId="lighteningChangeThreshold_A2"
                    callback="recomputeThreshold"/>
      </compoundCondition>
</contextEvent>

<contextEvent id="lighteningChangeThreshold3">
      <compoundCondition logicOperation="and">
       <simpleCondition contextVariable="LighteningConditions"
                        comparator="gt"value="0.6"/>
       <simpleCondition contextVariable="NumberOfClusters"
                        comparator ="lt" value="4"/>
       <action actionId="lighteningChangeThreshold_A2"
                    callback="recomputeThreshold ">
      </compoundCondition>
</contextEvent>

<contextEvent id="powerProfileChange1">
      <compoundCondition logicOperation="and">
       <simpleCondition contextVariable="PowerProfile"
                        comparator="eq" value="HIGH"/>
       <action actionId="lighteningChangeThreshold_A2"
                    callback ="recomputeThreshold ">
      </compoundCondition>
</contextEvent>

<contextEvent id="powerProfileChange2">
      <compoundCondition logicOperation="and">
       <simpleCondition contextVariable="PowerProfile"
                        comparator="eq"value="LOW"/>
       <action actionId="lighteningChangeThreshold_A2"
                    callback ="recomputeThreshold ">
      </compoundCondition>
</contextEvent>

<contextEvent id="powerProfileChange3">
      <compoundCondition logicOperation="and">
       <simpleCondition contextVariable="PowerProfile"
                        comparator="eq"value="MEDIUM"/>
       <action actionId="lighteningChangeThreshold_A2"
                    callback="recomputeThreshold ">
      </compoundCondition>
</contextEvent>
```

Codigo D.1: Pseudo código do callback usado na aplicação AISS.

```
-- image threshold callback implementation
function recomputeThreshold(lightenigCondition,
       numberOfClusters)

      -- image sample used to tunne the threshold filters
      -- according the current lighteningConditions
      imageSample = getImageSample (lightenigCondition)

      -- function tunneYCbCr, RGB and YUV return the
```

```
      -- results of applying the threshold filters to
      -- the image in the current conditions. The best
      -- of the current filters will be used.
      ycbcrThresholds,nClustersYCbCr = tunneYCbCr(imageSample)
      rgbThresholds,nClustersRGB = tunneRGB(imageSample)
      yuvThresholds,nClustersYUV = tunneYUV(imageSample)

      -- the optimizer will run to search the optimal
      -- architecture for the current power profile.

      arch = optimizer.getArchitecture(powerProfile)

      if (nClustersYCbCr >= acceptableNumber) then
       arch.setComponentParam("ColorTransformation",
            "transformation", "YCBCR" )
       arch.setComponentParam("thresholdFilter",
            "thresholds", ycbcrThresholds)
       arch.setComponentParam("thresholdFilter",
            "format", "YCBCR")
      else if (nClustersRGB >= acceptableNumber) then
       arch.setComponentParam("ColorTransformation",
            "transformation", "NULL")
       arch.setComponentParam("thresholdFilter",
            "thresholds", rgbThresholds)
       arch.setComponentParam("thresholdFilter",
            "format", "RGB")
      else if (nClustersYUV >= acceptableNumber) then
       arch.setComponentParam("ColorTransformation",
            "transformation", "YUV")
       arch.setComponentParam("thresholdFilter",
            "thresholds", yuvThresholds)
       arch.setComponentParam("thresholdFilter",
            "format", "YUV")
      end

      -- configure the optimal architecture by generating
      -- its files and sending it to the EmbeddedSystem
      -- if needed.
      configure ( arch )
end
```