Universidade Federal do Maranhão
Centro de Ciências Exatas e Tecnologia
Curso de Ciência da Computação

Michael Costa de Castro

Aplicação da Engenharia Dirigida por Modelos para Suportar a Computação em Nuvem: Ênfase em Tolerância a Falhas

São Luís
2015

Michael Costa de Castro

Aplicação da Engenharia Dirigida por Modelos para Suportar a Computação em Nuvem: Ênfase em Tolerância a Falhas

Monografia apresentada ao curso de Ciência da Computação da Universidade Federal do Maranhão, **como parte dos requisitos necessários** para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Dr. Denivaldo Cicero Pavão Lopes

Dr. em Ciência da Computação - UFMA

São Luís

2015

Castro, Michael Costa de

Aplicação da Engenharia Dirigida por Modelos para Suportar a Computação em Nuvem: Ênfase em Tolerância a Falhas / Michael Costa de Castro - São Luís, 2015.

74.f

Orientador: Prof. Dr. Denivaldo Cicero Pavão Lopes

Monografia (Graduação) - Universidade Federal do Maranhão, Curso de Ciência da Computação, 2015.

1.Computação em Nuvem 2. Engenharia Dirigida por Modelos 3. Tolerância a Falhas. I.Título.

CDU 004.41:62

Michael Costa de Castro

Aplicação da Engenharia Dirigida por Modelos para Suportar a Computação em Nuvem: Ênfase em Tolerância a Falhas

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal do Maranhão, **como parte dos requisitos necessários** para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 8 de julho de 2015

BANCA EXAMINADORA

Dr. Denivaldo Cicero Pavão Lopes

Dr. em Ciência da Computação - UFMA

Dr. Geraldo Braz Júnior

Dr. em Ciência da Computação - UFMA

Dr. Samyr Béliche Vale

Dr. em Ciência da Computação - UFMA

Ph.D. Zair Abdelouahab

Ph.D. em Ciência da Computação - UFMA

Aos meu pais, familiares e amigos

Resumo

A Engenharia Dirigida por Modelos (*MDE - Model Driven Engineering*) permite o gerenciamento da complexidade de desenvolvimento, manutenção e evolução de sistemas computacionais, baseando-se em modelos e transformações entre modelos. Este trabalho apresenta uma abordagem baseada na Engenharia Dirigida por Modelos (MDE) para suportar a Tolerância a Falhas nos sistemas de software em ambientes de Computação em Nuvem. Um *framework*, uma metodologia e metamodelos são propostos para suportar tolerância a falhas. Metamodelos específicos para plataformas de Computação em Nuvem são propostos. Um exemplo ilustrativo é realizado para uma melhor compreensão da abordagem e utilização do *framework*. Uma análise do *framework* e suas futuras melhorias são apresentadas.

Palavras-chave: Computação em Nuvem, Engenharia Dirigida por Modelos, Tolerância a Falhas.

Abstract

Model Driven Engineering (MDE) allows the management of development, maintenance and evolution of complex computational systems, based on models and transformations between models. This work presents an approach based on Model Driven Engineering to support Fault Tolerance in software systems for cloud computing environments. A framework, a methodology and metamodels are proposed to support fault tolerance. Specific metamodel for Computing Cloud platforms are proposed. An illustrative example is performed for a better understanding of the approach and use the framework. A framework analysis and future improvements are presented.

Keywords: Cloud Computing, Fault Tolerance, MDE.

Agradecimentos

Em primeiro lugar agradeço a Deus por estar sempre ao meu lado.

Aos meus pais Raimunda e Manoel e aos meus Irmãos Michele, Rafael e Carlon.

Ao meu orientador, Dr. Denivaldo Cicero Pavão Lopes, pela orientação na monografia e em outros projetos.

Aos meus colegas da graduação, em especial, a Regina Célia, Sasha Nícolas, Thiago Mendes, André Jorge, Aitan Viegas, Luis Felipe, Ruy Guilherme e Valéria Monteiro por toda a convivência e companheirismo durante a graduação.

Aos integrantes e ex-integrantes do LESERC, em especial Jéssica Bassani, Eduardo Devidson, Paulo Jansey, Pablo Matos, Wesley Leite, Steve Ataky, Gustavo Dominices, Narana Pinheiro, Aline Luiza, Amanda Serra e Gerson Lobato.

A minha namorada, Dayana Mendonça, pela ajuda, companheirismo e compreensão durante todo esse processo.

Aos meus amigos que estiveram presentes em toda a graduação: Eliziane, Jairo, Cacio, Karllyane, Leidiane, Elidiane, Jairiane, Wetila, Mônica, Denilson, Auricelia e Saniele.

Ao PIBITI e a FAPEMA pelo apoio e as oportunidades concedidas ao longo da graduação.

Ao BRAFITEC e a CAPES pela o oportunidade na participação do intercâmbio Brasil-França.

Ao professor João Viana e ao Brice Duhamel pela orientação durante o intercâmbio na França.

A todos que de alguma forma contribuíram com esse trabalho.

" Aprender é a única coisa de que a mente nunca se cansa, nunca tem medo e nunca se arrepende ".

Leonardo da Vinci

Sumário

Lista de Figuras

1 Introdução

Atualmente, os software estão presentes em todas as áreas, isso requer uma maior capacidade de processamento e armazenamento das máquinas. Este fator leva a um aumento da complexidade dos sistemas de informação, surgindo assim, muitas dificuldades no desenvolvimento, no gerenciamento e na manutenção dos mesmos.

A Internet abre uma larga gama de possibilidades e suporte para inovações para o mundo da informática. A Computação em Nuvem *(Cloud Computing)*[33] é uma das possibilidades que está começando a ser muito utilizada, nela todo o processamento e armazenamento pode ser realizado nos servidores remotos, sendo acessível de qualquer parte do mundo através da Internet. Desse modo, a Computação em Nuvem possui muitas vantagens, tais como: melhor aproveitamento dos recursos de hardware, economia e flexibilidade, existindo ainda muitos desafios a serem vencidos. Problemas de interoperabilidade entre as plataformas é uma grande questão a ser resolvida, pois um sistema feito em uma determinada plataforma não pode ser migrado facilmente para outra plataforma.

Existem muitos paradigmas que tentam resolver esses problemas, dentre eles aparece a Engenharia Dirigida por Modelos *(Model Driven Engineering - MDE)*, que possui o modelo como o centro de todo o processo de desenvolvimento. Algumas abordagens MDE são a *Arquitetura Dirigida por Modelos (Model Driven Architecture - MDA)*[26] e o *Eclipse Modeling Framework (EMF)*[34]. Com essas abordagens é possível criar uma lógica independente e depois transformá-la para modelos da plataforma específica.

A partir do momento em que um sistema é desenvolvido, este está propenso a falhas. Não é de hoje que ocorrem falhas em sistemas e estas são praticamente inevitáveis. No entanto, as suas consequências podem ser evitadas através de algumas técnicas tais como backup dos dados ou a redundância de equipamentos, cabendo ao desenvolvedor avaliar qual a melhor técnica a ser utilizada[36].

Falhas podem ocorrer tanto no nível de hardware quanto no nível de software e suas principais causas são problemas de especificação, de implementação, componentes defeituosos, além de distúrbios externos[36]. Por muito tempo esse problema atingiu os softwares instalados localmente. Com a Computação em Nuvem não é diferente, as falhas

podem acontecer e gerar danos significativos.

Esforços tem sido feitos para diminuir essas falhas, mas muitas dificuldades são encontradas pois existem muitas plataformas diferentes.

1.1 Problemática

Durante as pesquisas realizadas no campo de Tolerância a Falhas para a Computação em Nuvem, percebeu-se a falta de interoperabilidade entre as plataformas de nuvem[17][22] e a escassez de abordagens de prevenção a falhas voltadas para a Computação em Nuvem[14][16].

Este trabalho propõe uma abordagem baseada em MDE para suportar a interoperabilidade entre as plataformas de nuvem e oferecer uma abordagem de Tolerância a Falhas para a criação e gerenciamento de aplicações.

1.2 Motivação

Atualmente, existem diversas plataformas de nuvem, mas não há um padrão entre os provedores de serviços. Muitas pesquisas[17][22][14][16] têm sido feitas tentando buscar a melhor forma de alcançar a interoperabilidade entre plataformas. A MDE vem se mostrando uma ferramenta eficaz no caminho para conseguir a interoperabilidade.

As técnicas de tolerância a falhas são importantes, desde os sistemas mais simples aos grandes e complexos. Em virtude disso, é essencial que esteja presente nos sistemas modernos, principalmente nos sistemas desenvolvidos nas plataformas de nuvem. Olhando para este tema, percebe-se a necessidade de existir a interoperabilidade de sistemas tolerantes a falhas para a computação em nuvem.

1.3 Solução Proposta

Utilizar MDE para a criação de uma metodologia, metamodelos e *framework* que permita o desenvolvimento de software tolerante a falhas na Computação em Nuvem, permitindo a utilização desta solução em diferentes plataformas.

O *framework* proposto permite a implementação de sistemas tolerantes a falhas para plataformas na nuvem do tipo SaaS (*Software as a Service*), facilitando as transferências dos mesmos para outros provedores, diminuindo os custos e aumentando a velocidade.

1.4 Objetivos

O objetivo geral do trabalho é propor uma abordagem baseada em Engenharia Dirigida por Modelos para suportar a computação em nuvem focando em tolerância a falhas.

1.4.1 Objetivos Específicos

Os objetivos específicos deste trabalho consistem nos seguintes itens:

- Estudar e desenvolver conceitos relacionados a Engenharia Dirigida por Modelos;
- Desenvolver uma abordagem MDE para a computação em nuvem focando em tolerância a falhas;
- Propor um *framework* baseado em MDE para suportar a computação em nuvem focando o aspecto tolerância a falhas.

1.5 Metodologia

A metodologia empregada para a realização desde trabalho pode ser listada a seguir:

1. Levantamento e análise bibliográfica sobre conceitos e trabalhos científicos abordando:
   - Engenharia Dirigida por Modelos;
   - Linguagem de transformação de modelos;
   - Metamodelagem, *Eclipse Modeling Framework (EMF)* e *Graphical Modeling Framework (GMF)*;

   - Computação em Nuvem e Grade Computacional;
   - Qualidade em desenvolvimento tecnológico, frisando Tolerância a Falhas.

2. Estudo sobre as topologias de computação em nuvem para determinar quais características devem estar presente na construção do *framework*;
3. Proposta de metamodelos para a Computação em Nuvem;
4. Proposta de um *framework* de Tolerância a Falhas para a Computação em Nuvem;
5. Criação de definições de transformação utilizando a linguagem de transformação *Atlas Transformation Language (ATL)*;
6. Proposta de criação de uma *DSL (Domain Specific Language)* para computação em nuvem baseada em Engenharia Dirigida por Modelos.

1.6 Apresentação dos Capítulos

Este trabalho de conclusão está dividido em seis capítulos, da seguinte forma:

O primeiro Capítulo apresenta e introduz o contexto atual do tema do trabalho, explicando a escolha do assunto, a problemática, a motivação e a solução proposta para o mesmo. Ainda no capítulo, os objetivos e a metodologia empregado no desenvolvimento do trabalho são apresentados.

O segundo Capítulo trata da fundamentação teórica, mostrando os conceitos necessários para o desenvolvimento e o entendimento desse trabalho. Nele são explicadas tecnologias, tais como: *Engenharia Dirigida por Modelos (MDE)*, *Arquitetura Dirigida por Modelos (MDA)*, *Eclipse Modeling Framework (EMF)*. As Linguagens de Transformação e característica do processo da MDA são apresentadas, em específico: a *Atlas Transformation Language (ATL)* e *MOF/QVT*. No fim do Capítulo, a Computação em Nuvem e as Técnicas de Tolerância a Falhas, essenciais para esta pesquisa, são abordados em detalhes.

No terceiro Capítulo, o Estado da Arte com os temas relacionados às pesquisas mais relevantes são explicados de forma resumida. Elas serviram de base bibliográfica para este trabalho, com destaque para: Computação em Nuvem, MDE e Tolerância a Falhas.

No quarto Capítulo, a abordagem de Tolerância a Falhas para a Computação em Nuvem é apresentada. Os metamodelos e a concepção básica do *framework* são explicadas em detalhes. As definições de transformação, feitas em ATL, são exibidas.

O quinto Capítulo mostra o *framework* proposto implementado no Eclipse, um exemplo ilustrativo é apresentado para um melhor entendimento da ferramenta.

No sexto e último Capítulo, as conclusões, os objetivos atingidos, as limitações e os trabalhos futuros são apresentados e analisados.

2 Fundamentação Teórica

O desenvolvimento desta monografia utilizou uma série de conceitos, ferramentas e tecnologias. Deste modo, é necessária a apresentação desses termos para o melhor entendimento possível dessa pesquisa.

Este Capítulo apresenta os conceitos de *Model Driven Engineering (MDE)* e do *framework Model Driven Architecture (MDA)* baseado na abordagem MDE. O *Eclipse Modeling Framework (EMF)* também é apresentado, assim como a linguagem de transformação *Atlas Transformation Language (ATL)* e *MOF/QVT*. A Computação em Nuvem e a Tolerância a Falhas são explicados com mais detalhes.

2.1 Engenharia Dirigida por Modelos (MDE)

A Engenharia Dirigida por Modelos (*MDE-Model Driven Engineering*) é uma abordagem que possui modelos como foco principal e presente em todas as etapas do processo de desenvolvimento de software, com o objetivo de prover benefícios como o gerenciamento da complexidade deste, a harmonização entre as diversas tecnologias existentes, redução de custos, diminuição de tempo no processo de criação, manutenção e/ou adaptação do software e aumento da qualidade do mesmo [31].

Com MDE, o desenvolvimento pode se concentrar na criação de modelos. Alguns conceitos são importantes para a utilização da abordagem MDE:

- Modelo: *"é a descrição ou especificação de um sistema e seu ambiente para um determinado propósito"*[26] que pode ser representado como uma combinação de desenhos e textos. Em [15], modelo *"é a descrição de um sistema ou parte dele escrito em uma linguagem bem definida sendo que essa linguagem possui um formato preciso (sintaxe) e um significado (semântica) que pode ser interpretado de forma automatizada por computador"*;

- Metamodelo: em [26], a definição de metamodelo é "um modelo que define a linguagem para expressar um modelo";

- Linguagem de Modelagem: *"Uma linguagem de modelagem é uma especificação formal bem definida que contém os elementos de base para construir modelos"* [19] .

2.1.1 Model Driven Architecture (MDA)

Arquitetura Dirigida por Modelos (MDA) é uma proposta definida pela OMG para desenvolvimento de software, onde, seguindo a abordagem MDE, os modelos são o foco principal do processo de desenvolvimento de software[25] [26].

De acordo a abordagem MDA, os modelos são divididos em dois tipos, Modelo Independente de Plataforma (PIM) e Modelo Específico de Plataforma (PSM). Os Modelos Independentes de Plataforma são modelos genéricos que não dependem da plataforma onde o software será implementado. Os Modelos Específicos de Plataforma possuem informações específicas da plataforma de implementação[26].

MDA se baseia em sucessivos processos de transformação, para que isso ocorra é necessário um Motor de Transformação e Definições de Transformação. Motor de Transformação é o responsável por realizar a transformação para um modelo específico de acordo com regras de transformação [15]. Definição de Transformação *"é o conjunto de regras de transformação que juntas descrevem como um modelo em uma linguagem fonte pode ser transformado em um modelo de uma linguagem alvo."* [15]. As Definições de Transformação são escritas em uma Linguagem de Transformação como a ATL e MOF/QVT, explicadas na Seção 2.1.6.

2.1.2 Eclipse Modeling Framework (EMF)

Um *framework* é um conjunto de funções ou classes (relacionadas ou não) para alcançar um ou mais objetivos, tais como:

- Simplificação de uso;
- Consistência na interface;
- Melhoria da funcionalidade básica[6].

Um framework é uma estrutura genérica que pode ser ampliada para criar um subsistema ou uma aplicação mais específica. É implementado em um conjunto de classes

abstratas ou concretas [32].

O EMF (*Eclipse Modeling Framework*) é um *framework* definido pelo Projeto Eclipse, com o objetivo de dispor ferramentas de modelagem e a geração de código para desenvolvimento de software segundo a abordagem MDE[34].

O EMF é composto por três peças fundamentais[9]:

- EMF - O EMF inclui o metamodelo Ecore que é usado para descrever modelos, e suporta a execução, edição e armazenamento de modelos que podem ser importados/exportados no padrão XMI (*XML Metadata Interchange*);
- EMF.Edit - O *framwork* EMF.Edit oferece suporte para a construção de editores de modelos;
- EMF.Codegen - O EMF Code Generator é capaz de gerar o necessário para a construção de um editor de modelo EMF.

No EMF, três níveis de geração de código são suportados[9]:

- Model - Fornece interfaces Java e classes de implementação para todas as classes no modelo;
- Adapters - Gera classes de implementação que se adaptam as classes do modelo para edição e exibição;
- Editor - Produz um editor devidamente estruturado que está de acordo com o estilo recomendado para editores do Eclipse EMF e serve como um ponto de partida para começar a personalização.

2.1.3 Software Factories

Segundo [10], *Software Factories* é considerado um ambiente de desenvolvimento configurado para suportar o desenvolvimento de um tipo específico de aplicação. *Software Factories* é considerado a evolução natural dos métodos e práticas de desenvolvimento de sistemas. Ele possui três conceitos chaves: Um *Software Factory Schema*, um *Software Factory Template* e um Ambiente de Desenvolvimento Extensível.

Software Factory Schema é um documento usado para categorizar e resumir os artefatos usados na construção e manutenção de um sistema. O *Software Factory Template* são as DSL, os padrões, as estruturas e as ferramentas que foram descritas no *Software Factory Schema* e o Ambiente de Desenvolvimento Extensível é o local onde o *Software Factory Template* pode ser configurado e manipulado [10][30].

2.1.4 Unified Modeling Language (UML)

A *Unified Modeling Language* ou Linguagem de Modelagem Unificada é uma linguagem gráfica utilizada para modelar sistemas de softwares. A UML é orientada a objetos e tornou-se um padrão de grande aceitação na engenharia de software[11][29].

A linguagem UML é composta por diagramas, os principais são[11]:

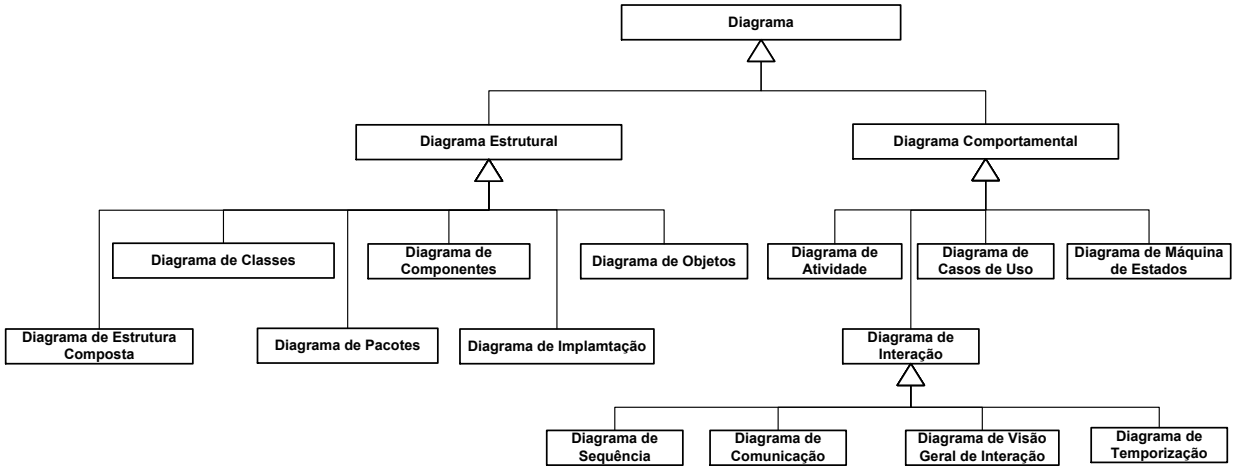
- Diagrama de Caso de Uso: tem como objetivo representar a ideia geral do sistema, identificando os atores e as funcionalidades de cada um;
- Diagramas de Classes: informa as classes, os atributos e os métodos utilizados pelo sistema, além dos seus relacionamentos;
- Diagrama de Objetos: possui os valores dos objetos, definidos no diagrama de classes, em execução no sistema;
- Diagrama de Sequência: mostra as trocas de mensagens entre os objetos dentro de um processo, se baseia em uma funcionalidade do caso de uso;
- Diagrama de Atividade: descreve o fluxo de controle de uma atividade, detalhando os passos para a conclusão da mesma.

Os diagramas UML podem ser classificados como diagramas estruturais e comportamentais, ainda há uma terceira divisão no diagrama comportamental, o diagrama de Interação. A Figura 2.1 mostra a divisão dos diagramas na UML em visões estruturais e comportamentais.

2.1.5 Linguagens Específicas de Domínio (DSL)

DSL são linguagens que surgiram para um conjunto específico de tarefas. Uma DSL permite que um domínio específico tenha o seu escopo reduzido através da abstração

Figura 2.1: Diagramas UML



Fonte: Guedes[11]

que a mesma providencia. Podem ser necessárias várias linguagens para esconder todo um problema, mas cada uma dela resolve uma parte e esconde a mesma atrás de uma abstração mais simples. Ela esconde a complexidade do código. A DSL não pode ser mais complicada do que o código que pretende substituir. A abstração deve ser inequívoca, mas com uma interface significativamente reduzida[8].

2.1.6 Linguagens de Transformação

Linguagem de transformação é uma linguagem que define operacionalmente a transformação de um modelo fonte em um modelo alvo. Ela manipula elementos de modelos, considerando os metamodelos utilizados na construção dos mesmos[18].

Existem várias linguagens de transformação, por exemplo: ATL (Atlas Transformation Language)[12], YATL (Yet Another Transformation Language, BOTL (Basic Object-oriented Transformation Language) e MOF/QVT (Query View Transformation).

2.1.6.1 Atlas Transformation Language (ATL)

A *Atlas Transformation Language* é uma linguagem de transformação de modelos e metamodelos. Com ela é possível criar regras de transformação que definam a correspondência entre um modelo fonte e um modelo alvo[12].

A linguagem de transformação ATL é uma linguagem híbrida, pois aceita tanto construções declarativas quanto construções imperativas. A parte declarativa da

linguagem é baseada em regras de combinação. Essas regras são compostas por padrões fonte e padrões alvo. Os padrões fonte são combinados com o modelo fonte e o padrão alvo é criado no modelo alvo para cada uma das combinações do modelo fonte. A navegação nos modelos é realizada através de expressões OCL. A execução de uma regra em uma combinação cria automaticamente ligações de rastreabilidade. As transformações são sempre executadas de forma unidirecional[12].

2.1.6.2 MOF/QVT

QVT (*Query View Transformation*) é a linguagem de transformação padrão da OMG (*Object Management Group*) que segue o padrão MOF (*Meta-Object Facility*). A QVT possui uma parte declarativa e outra imperativa[27].

A parte declarativa da linguagem é a responsável por realizar as transformações entre os modelos. A parte imperativa realiza os mapeamentos e pode ser considerada como uma extensão da parte declarativa, além disso, ela também é utilizada para adicionar funcionalidades de outras linguagens e bibliotecas que não fazem parte da QVT[27].

2.2 Computação em Nuvem

Computação em Nuvem (Cloud Computing) se refere a aplicações e serviços que executam em uma rede distribuída usando recursos virtualizados e acessados através da Internet[33]. Existem diferentes tipos de nuvem, tais como: Nuvem pública, Nuvem privada e Nuvem Híbrida.

A Computação em Nuvem se dividem em 3 principais tipologias:

- Infrastructure as a Service (IaaS): fornece máquinas virtuais, armazenamento e qualquer recurso de hardware que o cliente necessite;
- Platform as a Service (PaaS): fornece máquinas virtuais, sistemas operacionais, aplicações, serviços e framework de desenvolvimento;
- Software as a Service (SaaS): fornece um ambiente com aplicações, gerenciamento e interface com o usuário.

Atualmente, os principais fornecedores de serviços na Nuvem são: o Google, Azure Platform e Amazon Web Services.

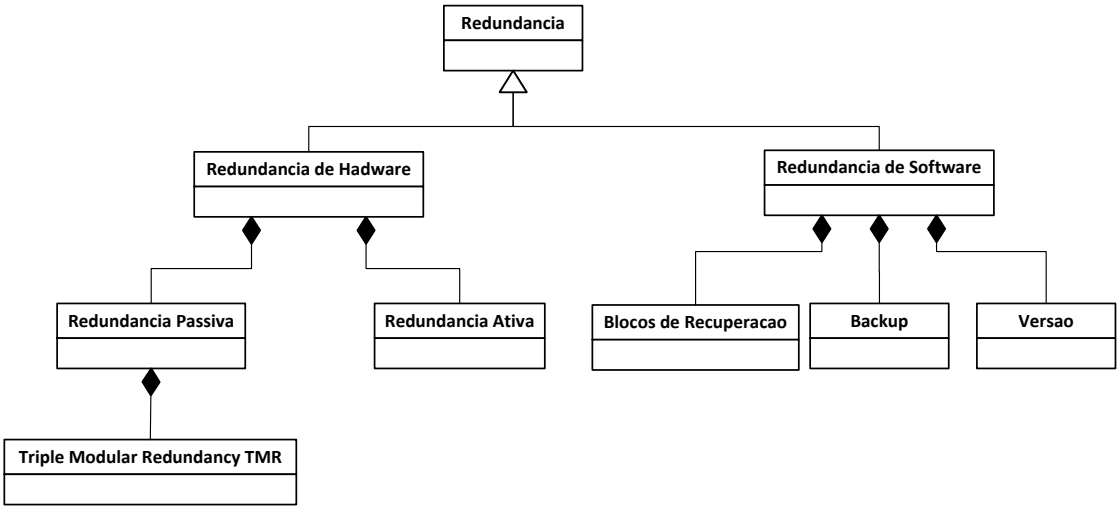
2.3 Tolerância a Falhas

Falhas são praticamente inevitáveis, mas as consequências das falhas, a interrupção no fornecimento do serviço e a perda de dados podem ser evitadas. Mesmo o mais simples dos sistemas devem implementar técnicas de tolerância a falhas, o que torna essencial para a sua disponibilidade e confiabilidade.

Para se entender o que é tolerância a falhas, os seguintes conceitos precisam ser mencionados:

- Falha, é um problema do hardware ou do software no nível físico;
- Erro, é a manifestação de uma falha no sistema e ocorrem no nível computacional;
- Defeito, é a manifestação de um erro em que o comportamento do sistema não corresponde ao esperado.

Figura 2.2: Técnicas de Tolerância a Falhas



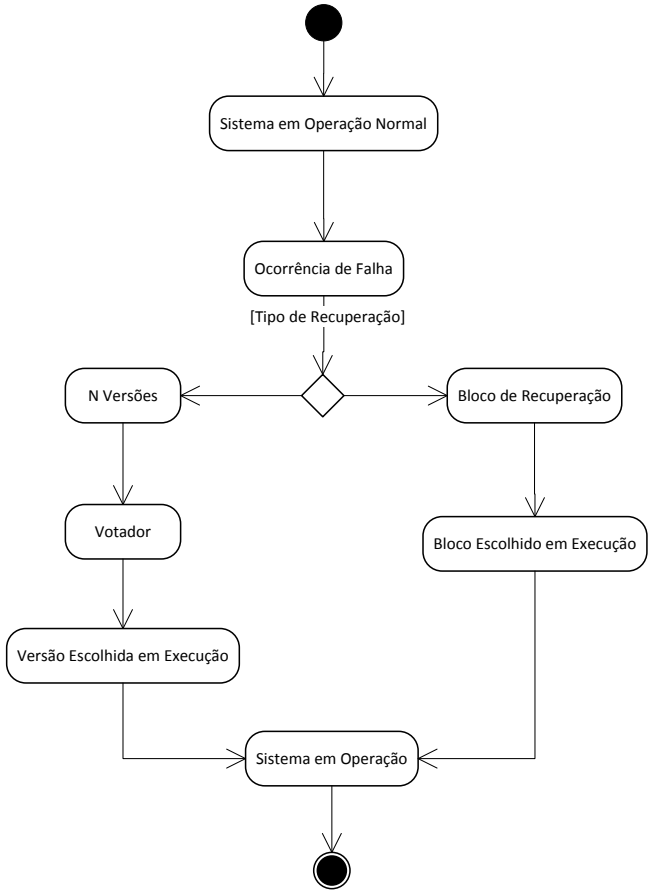
Fonte: Avizienis [4]

Outro conceito muito importante nessa área é o de dependabilidade, que indica a qualidade do serviço fornecido por um dado sistema e a confiança depositada no serviço fornecido. A dependabilidade é composta por algumas medidas:

- Disponibilidade: probabilidade do sistema estar operacional num instante de tempo determinado;
- Confiabilidade: capacidade de atender a especificação, dentro de condições definidas;
- Segurança (operacional): probabilidade do sistema ou estar operacional e executar sua função corretamente ou descontinuar suas funções de forma a não provocar dano a outros sistemas ou pessoas que dele dependam;
- Segurança (proteção): proteção contra falhas maliciosas.

Tolerância a Falhas pode ser entendido como um conjunto de técnicas para detectar, mascarar e tolerar falhas em um sistema com o objetivo de alcançar a dependabilidade.

Figura 2.3: Comportamento de Sistema de Software na ocorrência de uma falha



Fonte: Avizienis[4]

As técnicas de tolerância a falhas podem ser divididas em hardware e de software, todas baseadas, principalmente, em redundância. Em hardware pode existir a

redundância passiva, em que os elementos redundantes são usados para mascarar falhas, e redundância ativa, que emprega técnicas de detecção, localização e recuperação, sem mascaramento de falhas, a *triple modular redundancy* (TMR), redundância modular tripla, é uma das técnicas mais conhecidas de tolerância a falhas em hardware. Em falhas de software, existem as técnicas do backup de dados, blocos de recuperação e baseada em versões do software[4][36] . A Figura 2.2 mostra essas técnicas.

As técnicas de tolerância a falhas podem ser usadas durante o desenvolvimento de um sistema ou após a sua instalação. A Figura 2.3 é um exemplo que mostra um diagrama de atividades para a escolha da técnica que será usada para a correção da falha em software. O exemplo mostra um sistema em operação normal e ocorre uma falha, nesse ponto um tipo de recuperação é escolhida. Se a Técnica de N-Versões for escolhida, um algoritmo Votador irá decidir qual a melhor versão a ser usada para a falha, e em seguida, executar a versão escolhida [4]. Se a Técnica de Recuperação de Blocos é utilizada, o primeiro bloco que retornar um resultado satisfatório é escolhido para a execução[36].

2.4 Síntese

Este capítulo buscou explicar de forma resumida as tecnologias e os conceitos utilizados no decorrer deste trabalho.

Com relação a Tolerância a Falhas, os conceitos e as técnicas utilizadas neste trabalho foram apresentadas. Técnicas que utilizam redundância, como *Backup* e *Versionamento* são explicadas. Conceitos de Segurança, Confiabiliadade e Disponibilidade foram explicados, pois são de grande importância para o entendimento do tema.

A MDE também foi explicada. As definições de MDA e seu processo de transformação foram detalhados e conceitualizados. Os benefícios e o funcionamento do *framework* EMF foram apresentados.

As linguagens de transformação, essenciais no processo MDA, foram apresentadas como a linguagem MOF/QVT e a ATL, utilizada neste trabalho.

3 Estado da Arte

Este capítulo tem por objetivo apresentar de forma sucinta os principais estudos realizados nas áreas de MDE, Computação em Nuvem e Tolerância a Falhas, assim como, os conceitos, a arquitetura e os desafios que a computação em nuvem apresenta. Desta forma, algumas abordagens utilizando técnicas de Tolerância a Falhas, Computação em Nuvem e MDE são descritas.

A pesquisas relacionadas foram analisadas e serviram de base para este estudo.

3.1 MDE

Várias pesquisa mostram a utilização da MDE em diferentes áreas e situações, considerada por muitos autores uma abordagem eficaz para suportar soluções diante da grande variedade tecnológica existente.

Model Driven Engineering Approach Based on Aspects for High Speed Scientific X-rays Cameras [7]

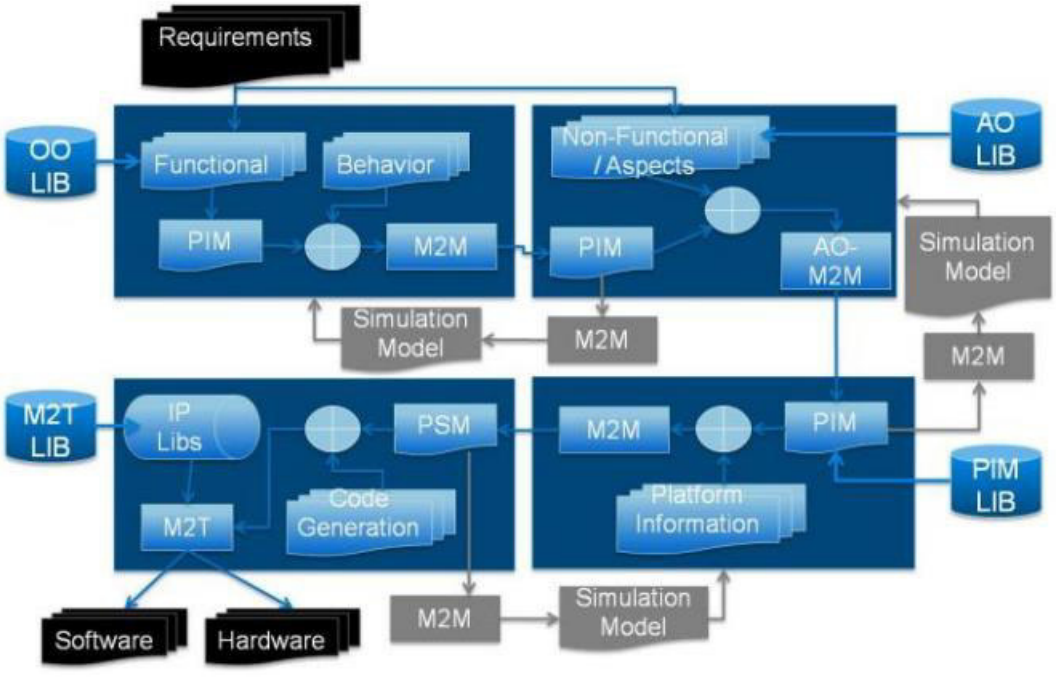
No trabalho realizado em [7], a metodologia chamada HIPAO é apresentada, Doering et al.[7] mostram como é possível criar algoritmos de processamento de imagens independente da plataforma de Hardware, baseadas em Engenharia Dirigida por Modelos.

A metodologia HIPAO visa diminuir os esforços de modelagem e reduzir o tempo de desenvolvimento. O método começa com a identificação e coleta dos requisitos, modelagem do sistema, inclusão das informações dos sistemas alvos dentro dos modelos, permitindo a transformação do modelo PIM para PSM, e a geração do código para a plataforma específica.

Este trabalho mostra o uso da MDE como uma metodologia viável para resolver e amenizar problemas que envolvam diversas plataformas. A Figura 3.1, retirada de Doering et al.[7], mostra a metodologia HIPAO de forma detalhada, funcionando da seguinte forma:

- Coleta de Requisitos: Este é o primeiro passo do processo, está representado na figura pelo bloco escuro no topo, que é modelado utilizando o diagrama SysML;
- Modelagem Orientada a Objetos: Cada requisito é mapeado para um bloco SysML, ao final dessa fase é obtido um modelo PIM que será a entrada da próxima fase;
- Modelagem Orientada a Aspecto: Nessa fase são adicionados comportamentos aos modelos;
- Projeto Exploração Espacial: Essa fase especifica o sistema de modelo que será usado em cada processo, é nela que o modelo PIM é transformado em PSM;
- Geração de Código: Esta é a etapa final da metodologia HIPAO, nela é onde ocorre a geração de código a partir do modelo PSM.

Figura 3.1: Visualização detalhada da metodologia HIPAO



Fonte: Doering et al.[7]

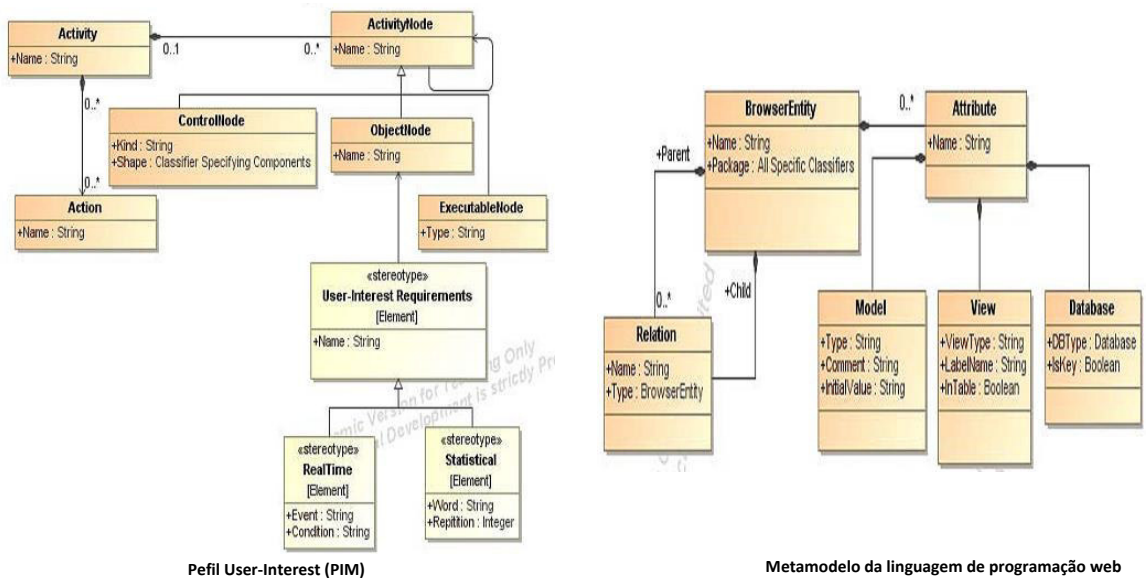
Enhanced Approach for Developing Web Applications Using Model Driven Architecture [2]

O trabalho apresentado em [2] propõe um mecanismo de aprimoramento do método de desenvolvimento de aplicações web, o WSDM *(Web Site Design Method)*, através da utilização da MDA. O mecanismo é denominado de WSDMDA.

O WSDM é uma abordagem que fornece uma completa metodologia para a construção de aplicações web. Abdalla et al [2]. propôs uma alteração na WSDM adicionando a MDA para a criação de aplicações web independente da plataforma de desenvolvimento.

A abordagem WSDMDA começa com a adição de um perfil chamado *User-Interest Aware* no diagrama de atividade do metamodelo UML (Figura 3.2), esse é o PIM. A Figura 3.2, retirada de [2], mostra o metamodelo da linguagem de programação web proposto que é o PSM da abordagem.

Figura 3.2: Abordagem WSDMDA



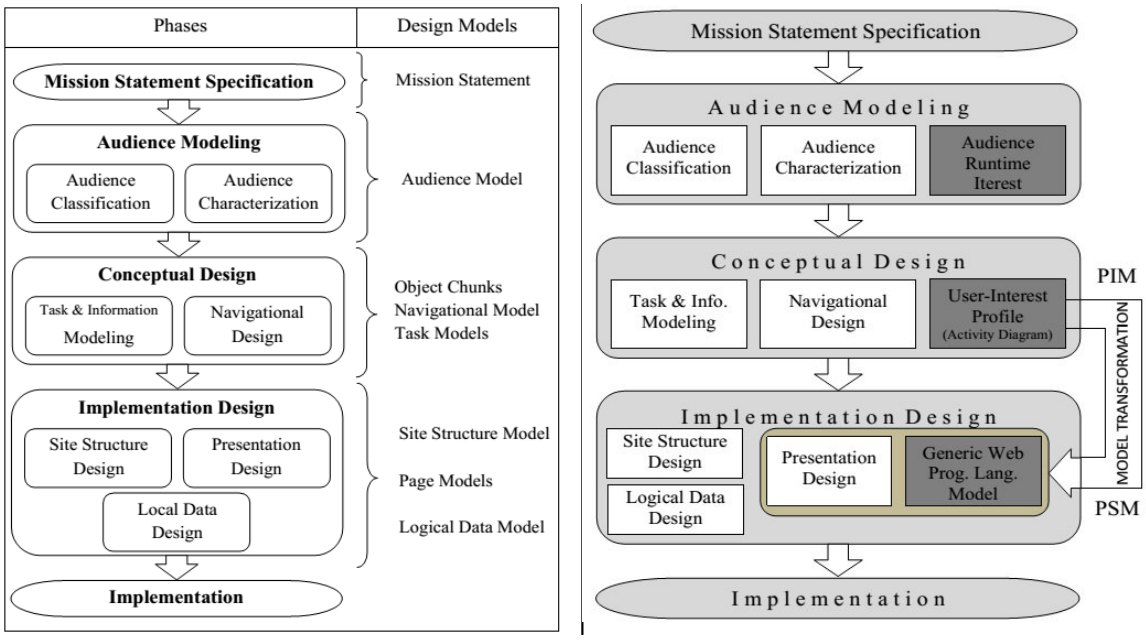
Fonte: Abdalla et al [2].

A Figura 3.3, do trabalho de [2], mostra a aplicação dos mecanismos propostos na Figura 3.2 dentro da abordagem WSDM. Nela, são mantidos todas as partes da WSDM (caixa branca) e adiciona novos componentes (caixa marrom). A parte mais importante da WSDMDA está no Projeto Conceitual e no Projeto de Desenvolvimento onde é executado a transformação do modelo PIM para PSM e depois a geração de código. As regras de

transformação foram geradas usando a linguagem QVT.

O mecanismo WSDMDA permite o método WSDM lidar com aplicações web dinâmicas em vez de somente estáticas. Na fase de implementação, o modelo de linguagem de programação genérica faz o WSDMDA rodar em diferentes plataformas (J2EE, PHP, or JSP), alcançando assim a flexibilidade desejada.

Figura 3.3: Comparação entre o WSDM[35] (esquerda) e o WSDMDA (direita)



Fonte: Abdalla et al [2].

3.2 Computação em Nuvem

A Computação em Nuvem tem sido alvo de várias pesquisas nos últimos anos, e apesar de seus benefícios, ainda existem desafios a serem vencidos. Esta seção visa descrever trabalhos relacionados a Computação em Nuvem, seus conceitos, sua arquitetura e seus desafios.

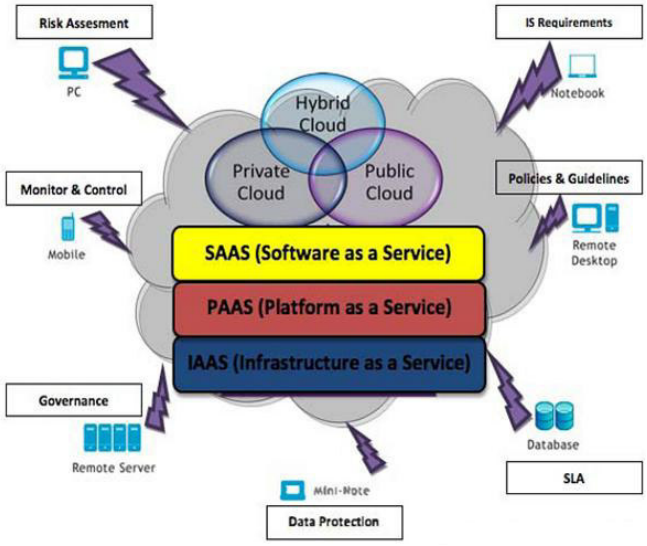
Cloud Computing - Concepts, Architecture and Challenges [13]

Jadeja et al[13] apresenta os conceitos, a arquitetura e os principais provedores de serviços da computação em nuvem são descritos. Segundo Jadeja et al.[13], ela é dividida em dois segmentos, o *front end* e o *back end*, em que os dois lados são conectados

através de uma rede, que normalmente é a Internet. O *front end* é o lado que o usuário vê enquanto que o *back end* é o sistema da nuvem.

Segunda a pesquisa, a computação em nuvem pode ser dividida em três camadas principais: Software as a Service (SaaS), Platform as a Service (PaaS) e Infrastruture as a Service (IaaS) como apresentado na Figura 3.4 de [13], tudo dependendo do objetivo do usuário. Para Jadeja et al[13], o primeiro passo ao desenvolver na nuvem é escolher que tipo deve ser utilizada, Nuvem Pública, Nuvem Privada ou Nuvem Híbrida.

Figura 3.4: Mapa da Computação em Nuvem



Fonte: Jadeja et al[13]

O artigo destaca as principais vantagens do uso da nuvem, como por exemplo: a facilidade no gerenciamento, a redução nos custos, está menos propício a falhas, boa gestão na ocorrência de erro e uma computação mais voltada ao meio ambiente. Apesar de todos os benefícios, a computação em nuvem também traz outras questões como a segurança e privacidade, uma vez que grandes quantidades de dados ficam de posse das empresas que prestam serviços na nuvem. O trabalho[13] conclui que a Computação em Nuvem gera uma infinita gama de possibilidades, mas ainda há muitas questões a serem resolvidas.

Toward Cloud Computing Reference Architecture: Cloud Service Management Perspective [3]

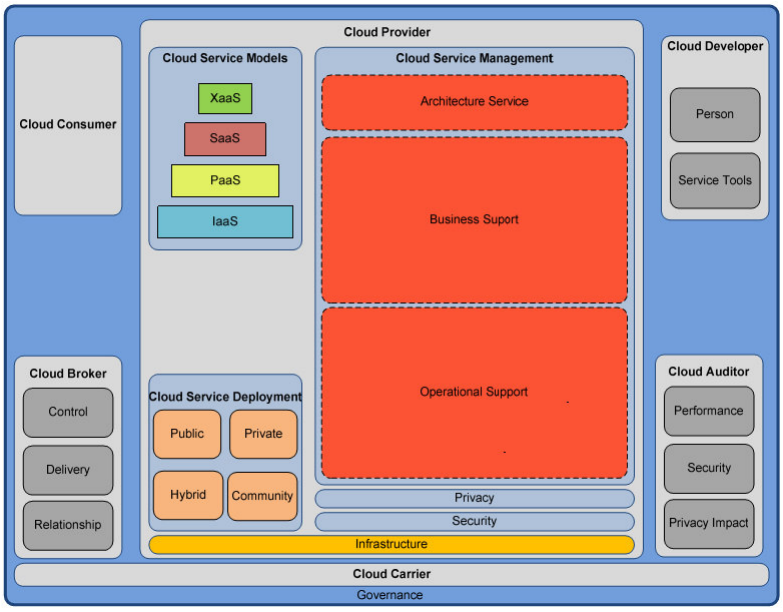
Juliandri et al [3] propõe uma nova arquitetura para a Computação em Nuvem, servindo como alternativa para os modelos já existentes. A Figura 3.5 mostra a arqui-

tetura proposta, possuindo seis principais atores, cada um com suas próprias atividades, requisitos e responsabilidades.

Os principais atores mostrado na Figura 3.5 possuem as seguintes funções:

- *Cloud Consumer*: representa os tipos e as expectativas dos consumidores da Nuvem;
- *Cloud Provider*: é o provedor de serviços da Nuvem, considerado o componente mais complicado;
- *Cloud Developer*: representa o responsável por desenvolver um serviço na nuvem, pode ser uma pessoal ou uma organização;
- *Cloud Broker*: é o agente negociador entre dois atores;
- *Cloud Auditor*: é o responsável por avaliar os serviços da nuvem;
- *Cloud Carrier*: representa a conexão entre os atores da nuvem.

Figura 3.5: Arquitetura Proposta



Fonte: Juliandri et al [3]

Um dos componentes mais importantes dessa arquitetura é o *Cloud Service Management* que é o responsável por oferecer os serviços com a melhor qualidade possível para os clientes da nuvem.

A principal contribuição desse trabalho foi, a criação de uma arquitetura simples e clara para o ambiente da computação em nuvem e por ser baseado em atores, permite um bom entendimento das atividades envolvidas.

3.3 Computação em Nuvem e MDE

Um dos maiores desafios da Computação em Nuvem é garantir a portabilidade entre os diferentes provedores de serviços existentes. Quando se possui a necessidade de migrar dados e aplicações de uma plataforma para outra, percebe-se que a interoperabilidade entre elas praticamente não existe. A MDE tem sido uma solução viável para esse problema. Diante disso, pesquisas [5][23] vêm sendo realizadas tentando criar uma abordagem MDE para Computação em Nuvem.

Towards a model-driven approach for promoting Cloud PaaS Portability [23]

Mattos Fortes et al[23] mostra a grande dificuldade encontrada por desenvolvedores na nuvem quando precisam levar dados de uma plataforma para outra, principalmente com relação a PaaS *(Platform as a Service)*. Isso ocorre devido a falta de padronização entre os diferentes provedores de nuvem.

Como é mostrado na Figura 3.6, a metodologia utilizada no trabalho funciona da seguinte forma:

1) Primeiro, alguns exemplos de aplicações foram criados para entender e analisar a PaaS;

2) Esses conceitos foram usados para criar uma linguagem de especificação, esta servindo de apoio à criação de modelos independentes de plataforma;

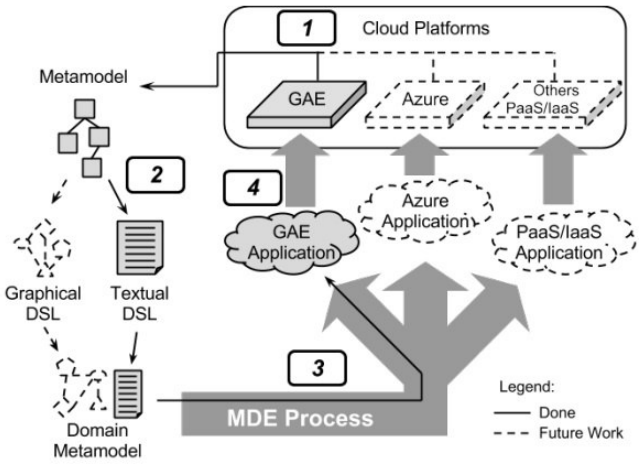
3) Baseadas nos exemplos estudados e na linguagem de especificação, algumas transformações para a geração automática do código foram realizadas;

4) Testes foram realizados para verificar a conformidade entre o código gerado e os requisitos da plataforma.

No trabalho [23] foram realizados testes na Google App Engine *(GAE)*, nuvem PaaS da Google. Uma aplicação de testes que realizava operações CRUD *(Create, Retrieve, Update, Delete)* em um banco de dados foi implementada, um metamodelo para as operações CRUD foi construído e transformações foram feita para a GAE.

O trabalho [23] demostra que o uso da MDE e a Computação em Nuvem juntas trazem uma série de benefícios para o desenvolvedor, como na produtividade, na

Figura 3.6: Metodologia de uma abordagem MDE para nuvem



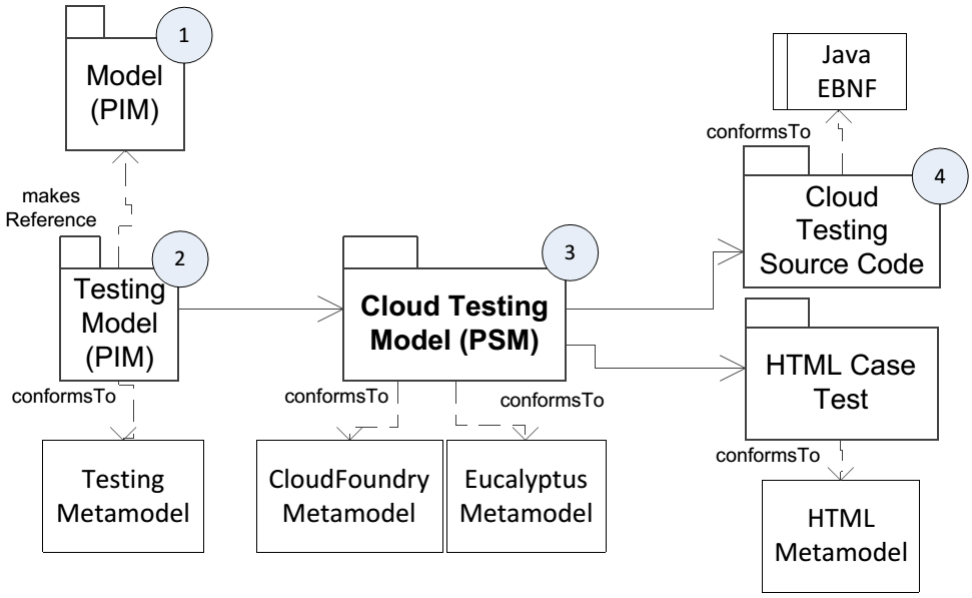
Fonte: Mattos Fortes et al[23]

manutenção e no reuso.

Model Driven Testing for Cloud Computing [24]

Oliveira et al[24] apresenta uma abordagem de criação de casos de testes para ambiente de computação em nuvem baseada em MDE. O objetivo é conseguir a interoperabilidade entre as diferentes plataforma de nuvem existentes.

Figura 3.7: Abordagem Proposta para Testes na Computação em Nuvem



Fonte: Oliveira et al[24]

A metodologia apresentada inclui a criação de metamodelos que permitem a geração dos casos de testes. A Figura 3.7, mostra a abordagem proposta, nela, primei-

ramente os metamodelos são criados, esses são os modelos independentes de plataforma (PIM), depois os modelos específicos (PSM) são gerados pelo modelo de transformação e por fim o código é gerado de acordo com o PSM.

As principais contribuições deste trabalho foram: a construção de uma metodologia e de metamodelos para a geração de casos de testes para a computação em nuvem utilizando MDE.

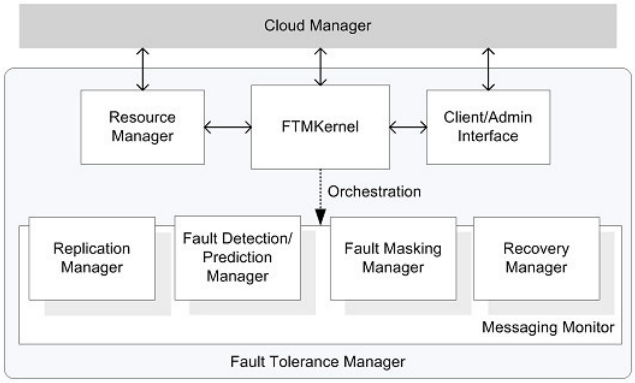
3.4 Tolerância a Falhas na Computação em Nuvem

A Computação em Nuvem traz muitos benefícios, mas a sua confiabilidade continua sendo uma grande preocupação entre os usuários [14]. Tentando amenizar esse problema, estudos têm sido feitos em busca de soluções utilizando técnicas de Tolerância a Falhas.

A Comprehensive Conceptual System-Level Approach to Fault Tolerance in Cloud Computing [14]

Jhawar et al[14] propõe um *framework* para o gerenciamento da Tolerância a Falhas na nuvem, visto que as técnicas até então existentes, possui certas limitações, seja a tolerância de falhas específicas ou fornecendo um único método de confiabilidade.

Figura 3.8: Arquitetura do Fault Tolerance Manager



Fonte: Jhawar et al[14]

A arquitetura do *framework*, denominado de *Fault Tolerance Manager* (FTM), é exibida na Figura 3.8 [14], cada componente funcionando da seguinte forma:

- **Replication Manager** (Gerenciador de Replicação): O FTM utiliza a redundância de aplicações, esse módulo gerencia as réplicas e quais serão utilizadas caso ocorra alguma falha no sistema;

- **Fault Detection/Prediction Manager** (Gerenciador de Detecção de Falhas): Esse componente gerencia a detecção de falhas, quando uma falha é detectada, uma notificação é enviada ao **FTMKernel** que invoca os serviços do **Masking Manager** e do **Recovery Manager**;

- **Fault Masking Manager** (Gerenciador de Mascaramento de Falhas): Esse módulo é o responsável por mascarar a ocorrência de falhas prevenindo que falhas resulte em erros, mantendo a aplicação em perfeito estado de execução;

- **Recovery Manager** (Gerenciador de Recuperação): Nesse componente estão os mecanismos para a retomada dos nós propenso a erros aos seus estados normais de execução;

- **Messaging Monitor**. Esse mecanismo, presente em todos os componentes do *framework*, disponibiliza a estrutura de comunicação necessária para a troca de mensagens entre os componentes;

- **Client/Admin Interface** (Interface do Cliente): Esse componente é usado para obter os requisitos do usuário, funcionando como a interface entre o usuário e o FTM;

- **FTMKernel**: É o componente central do *framework* que gerencia todos os mecanismos de confiabilidade;

- **Resource Manager** (Gerenciador de Recursos): Esse componente gerencia a alocação correta de recursos durante a ocorrência de falha, evitando um excesso de provisionamento.

A abordagem sugerida possui algumas vantagens como mais flexibilidade para os desenvolvedores, pois oculta parte dos detalhes de implementação das técnicas de confiabilidade. O *Framework* permite ao usuário captura sua própria perspectiva ganhando eficiente em nível de sistema.

Fault Tolerance in Cloud Computing [16]

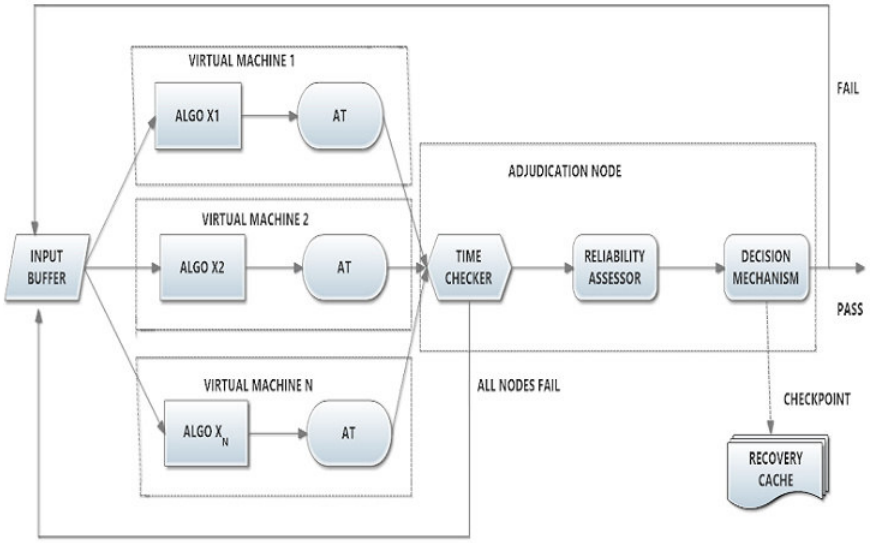
Lakshmi[16] propõe uma abordagem de Tolerância a Falhas para Computação em Nuvem que suporte aplicações de sistemas em tempo real.

A Figura 3.9 mostra a abordagem proposta em [16], nela, o modelo é baseado em nós (máquinas virtuais) que de acordo com sua confiabilidade podem ou não ser removidos. O modelo possui dois tipos de nós principais:

1) Conjunto de máquinas virtuais: onde fica a aplicação de tempo real e um teste de aceitação;

2) Nó de adjudicação: local onde se encontra o verificador de tempo, o assessor de confiabilidade e o mecanismo de decisão, esse mecanismo, fornece uma recuperação por avanço ou por retorno.

Figura 3.9: Tolerância a Falhas na Computação em Nuvem



Fonte: Lakshmi[16]

No modelo proposto, cada máquina virtual possui um algoritmo diferente e o módulo de teste de aceitação é o responsável pela verificação de cada nó, então o verificador de tempo checa o tempo de cada resultado e com base no assessor de confiabilidade cada nó tem sua confiabilidade calculada. Todos os nós são encaminhados ao mecanismo de decisão que seleciona o nó com maior confiabilidade.

Essa abordagem atende bem os requisitos para sistema de tempo real, pois possui o mecanismo de recuperação por avanço e comportamento dinâmico de configuração de confiabilidade levando a uma alta tolerância a falhas.

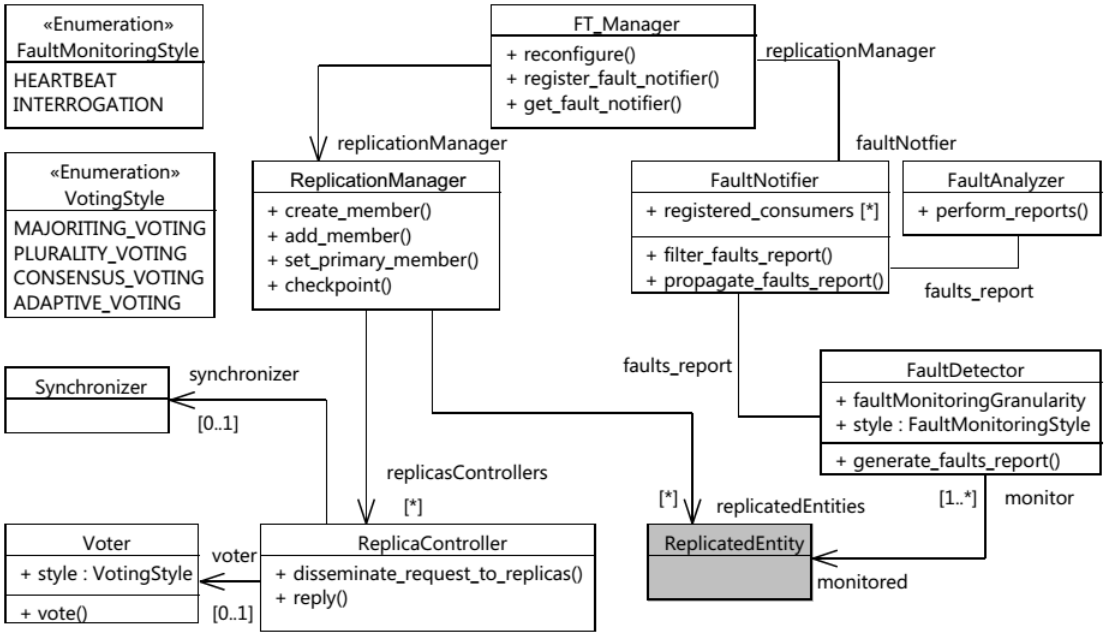
3.5 Tolerância a Falhas e MDE

Atualmente, devido a grande complexidade e diversidade de hardware e software disponíveis no mercado, fica cada vez mais custoso criar metodologia que abrangem todos. Vendo a necessidade de Tolerância a Falhas para qualquer aplicação diante da variedade de plataforma, muitos pesquisadores têm desenvolvido trabalhos nas áreas de Tolerância a Falhas e MDE.

A Model-Driven Engineering Framework for Fault Tolerance in Dependable Embedded Systems Design [37]

O trabalho apresentado propõe um *framework* baseado em MDE para sistemas embarcados com Tolerância a Falhas. Ele utiliza a técnica de tolerância a falhas por redundância, o foco está em modelar os artefatos para especificar a redundância tanto da aplicação quando da infraestrutura, permitindo a separação entre o modelo independente de plataforma e o modelo específico de plataforma. Primeiro, foi construído um metamodelo para abranger os conceitos da tolerância a falha, depois, um perfil UML voltado para a especificação e gerenciamento da redundância.

Figura 3.10: Metamodelo de gerenciamento DMF



Fonte: Ziani et al[37]

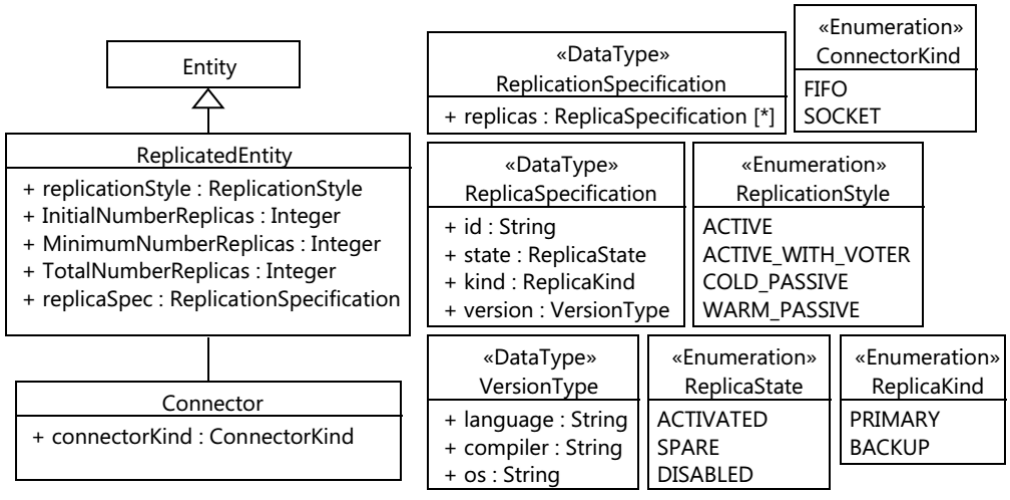
O *framework* denominado *Dependability Modeling Framework* (DMF), possui

duas partes:

1) O *DMF Management meta-model* é apresentado na Figura 3.10 [37], nessa parte está a definição da infraestrutura de tolerância a falhas. Nela existem dois tipos de gerenciamento, o de falhas e o de réplicas. O metamodelo define serviços de tolerância a falhas como: detecção de falhas, notificação de falhas, votação, checkpoint e sincronização. Para detectar quando um nó cai, mensagens são enviadas a partir do detector de falhas, enquanto que os componentes defeituosos são detectados por votação na requisições e resultados. Para tratar as falhas, cada componente possui o *FaultDetector* que assegura a detecção de alguma anomalia no nó, que produz um relatório e envia para o *FaultNotifier*, o *FaultAnalyzer* filtra os resultados que informa o *FT Manager* o estado de cada réplica;

2) O *DMF Specification meta-model* Figura 3.11 de [37]. Nessa parte são definidos os conceitos relacionados a especificação e a redundância.O componete *ReplicatedEntity* define a multiplicidade para representar o conjunto de réplicas e suas características. A especificação das réplicas fica no vetor DataType de *ReplicationSpecification*.

Figura 3.11: Metamodelo de especificação DMF



Fonte: Ziani et al[37]

Essa abordagem permite uma modelagem de tolerância a falhas em um alto nível de abstração, permitindo que uma mesma possa ser utilizada em diferentes plataformas.

3.6 Síntese

Este capítulo apresentou o Estado da Arte, que teve por objetivo descrever as pesquisas mais relevantes na área de Tolerância a Falhas, MDE e Computação em Nuvem. As abordagens e os conceitos vistos neste capítulo, serviram de base bibliográfica e deram embasamento para o desenvolvimento da abordagem proposta no capítulo a seguir.

Analisando estas pesquisas, pode-se chegar a conclusão de que a área da Computação em Nuvem é muito diversa, abrangendo muitos assuntos. Percebe-se uma tendência por buscar a interoperabilidade entre as plataformas existentes e que o uso da MDE é um caminho viável para que isso aconteça.

4 Uma Abordagem de Tolerância a Falhas em Computação em Nuvem

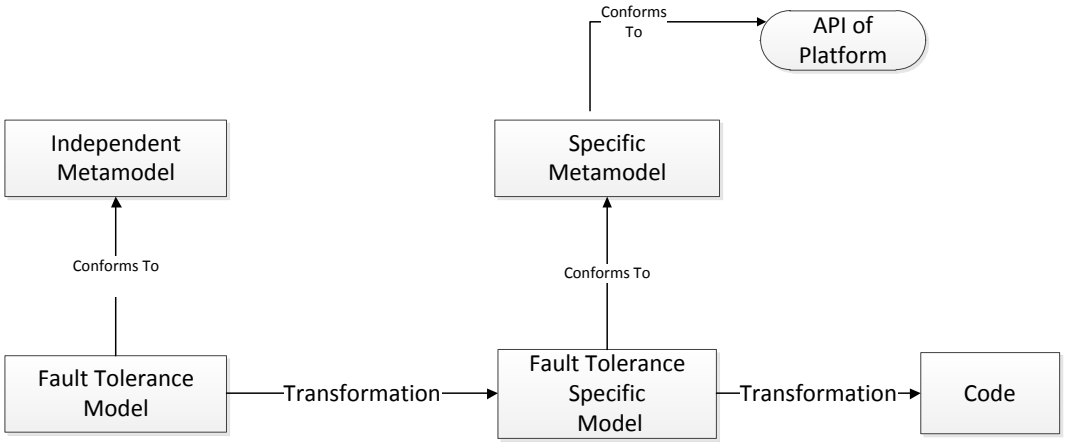
Este capítulo mostra a abordagem de Tolerância a Falhas em Computação em Nuvem.

O *framework* para o suporte a Tolerância a Falhas é apresentado. Os metamodelos propostos são explicados: o *FaultMetamodel*, o *CloudFault* e o *FaultCloudFoundry*. No final do capítulo, as definições de transformação são apresentadas.

4.1 Concepção Básica

A Figura 4.1 mostra a abordagem baseada em MDE para suportar a tolerância a falhas em computação em nuvem. Nela, pode-se perceber que são necessários quatro tipos de modelos:

Figura 4.1: Ideia básica da aplicação para o Suporte a Tolerância a Falhas



- **Metamodelo de Tolerância a Falhas**: Nele estão as regras para a criação de um modelo de Tolerância a Falhas, ele define quais técnicas podem ser utilizadas;
- **Modelo de Tolerância a Falhas**: Representa a modelagem do sistema com as Técnicas de Tolerância a Falhas, ele está de acordo com o Metamodelo de Tolerância

a Falhas;

- **Metamodelo Específico de Tolerância a Falhas**: Possui as regras para a criação do modelo de Tolerância a Falhas específico da plataforma da nuvem, ele leva em consideração a API (*Application Programming Interface*) da nuvem em que se pretende utilizar;

- **Modelo de Tolerância a Falhas Específico**: Representa o sistema com as Técnicas de Tolerância a Falhas específico para uma determinada nuvem, esse modelo está em conformidade com o Metamodelo de Tolerância a Falhas Específico da Nuvem.

A abordagem se baseia em uma sequência de transformações, característico do processo MDE, em que um modelo fonte é transformado em um modelo alvo. Para que essa abordagem funcione são necessários primeiramente dois tipos de metamodelos: um metamodelo de Tolerância a Falhas Independente da plataforma e outro metamodelo específico. Esses metamodelos são os que vão ditar as regras de criação dos modelos que serão transformados.

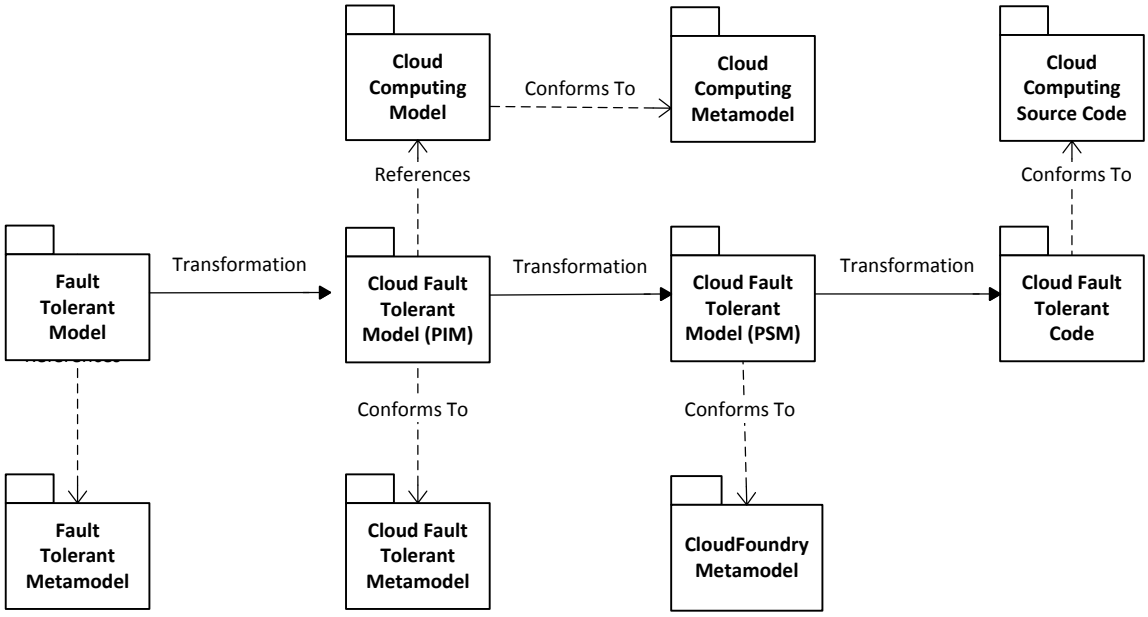
O Modelo de Tolerância a Falhas precisa ser criado pelo usuário da aplicação, ele é o ponto inicial do desenvolvimento do sistema utilizando a abordagem, este modelo deve estar de acordo com o seu Metamodelo de Tolerância a Falhas. O Modelo de Tolerância a Falhas Específico será criado semi-automaticamente pela aplicação através do processo de transformação. As regras de transformação se encarregam de fazer a correta correspondência do modelo fonte para o modelo alvo, assim como a transformação para o código.

4.2 Framework

Uma abordagem MDE para a computação em nuvem com tolerância a falhas é mostrada na Figura 4.2. Nela, um *framework* é fornecido e sua principal finalidade é suportar a interoperabilidade para que possa ser utilizado em diferentes plataformas de Computação em Nuvem. Essa abordagem é uma adaptação de [5].

O *framework* se concentra na criação de modelos visando alcançar a interoperabilidade, que é o principal objetivo. Ele permite que um sistema seja modelado junto

Figura 4.2: Abordagem MDE para computação em nuvem e tolerância a falhas (baseado e adaptado de [5])



com as técnicas de tolerância a falhas e pode trabalhar com diferentes tipos de nuvens como a IaaS, PaaS e SaaS.

O funcionamento da abordagem proposta começa com a criação de um modelo de Tolerância a Falhas, ainda sem as características de nuvem. Após sua criação, ocorre uma transformação para o Modelo de Tolerância a Falhas Independente da Plataforma, o PIM para a nuvem. Esse modelo está conforme as regras de criação estabelecidas no metamodelo de Tolerância a Falhas da nuvem, o qual é previamente oferecido pela abordagem. Após a criação do PIM, a sua transformação pode ser executada gerando um modelo PSM de Tolerância a Falhas para uma plataforma de nuvem específica, no caso, para a plataforma CloudFoundry. E por fim, esse PSM pode ser transformado em código fonte executável.

O *framework* trabalha com os metamodelos propostos na seção 4.3. O metamodelo *Cloud Computing Metamodel* foi utilizado do trabalho de [5]. Por ser uma abordagem MDE que possui como foco central os modelos, o *framework* consegue uma boa interoperabilidade entra as plataformas de computação em nuvem.

4.3 Metamodelos

Os metamodelos são de grande importância para a criação dos modelos e necessários para a construção e execução das regras de transformação. Essas regras utilizam os metamodelos fonte e alvo para definir as transformações entre os modelos[15]. Os metamodelos: *FaultMetamodel*, *CloudFault* e o *FaultCloudFoundry* são explicados detalhadamente.

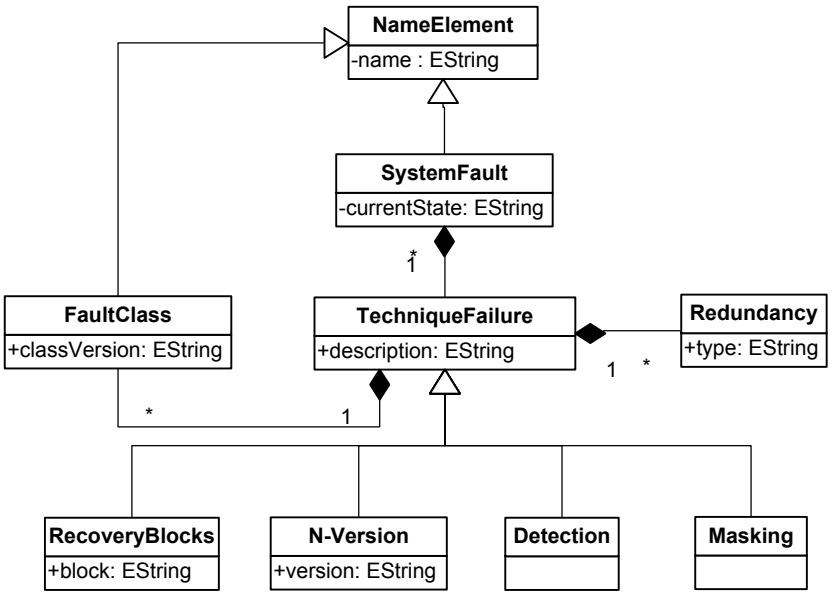
Metamodelo de Tolerância a Falhas Independente de Plataforma: *FaultMetamodel*

O metamodelo proposto na Figura 4.3 (*FaultMetamodel*) representa a metamodelagem de tolerância a falhas para sistemas independente de plataforma. Os principais elementos do metamodelo são:

- *NameElement* (Nome do Elemento): Representa a generalização das técnicas de tolerância a falhas;
- *TechniqueFailure* (Técnica de Falhas): Este é o principal elemento do metamodelo, pois é nele que é definido o tipo de técnica a ser aplicada. Ele possui o tipo de detecção que pode ser concorrente ou preemptiva;
- *SystemFault* (Sistema): Elemento que representa o sistema que utiliza a técnica de tolerância a falhas;
- *FaultClass* (Classe): Elemento que representa a classe do sistema onde é implementada a tolerância a falhas;
- *RecoveryBlocks* (Blocos de Recuperação): Elemento que representa a técnica de tolerância a falhas por blocos de recuperação, técnica utilizada em software;
- *Masking* (Mascaramento): Elemento que representa uma classe das técnicas de tolerância a falhas;
- *N-Version* (N-Versão): Elemento que representa o tipo de técnica de n-versões, usado em software;

- *Redundancy* (Redundância): Elemento que representa a redundância que é empregado em praticamente todas as técnicas de tolerância a falhas.

Figura 4.3: Metamodelo de Tolerância a Falhas



Pode-se observar no *FaultMetamodel*, que o elemento *Redundancy* está presente em todos os tipos de técnicas de tolerância a falha, o atributo *type* significa que cada técnica pode aplicar um tipo de redundância. Para cada técnica existe uma descrição (*description*) de como o algoritmo de prevenção a falhas funciona, esse algoritmo é definido pelo usuário, ficando a seu critério a escolha do melhor método para o sistema. O estado do sistema é descrito em *currentState* que é essencial para a aplicação da técnica de falha correta. *TechniqueFailure* contém uma referência da classe que implementa a técnica de falhas (*ClassFault*) que pertence ao elemento *SystemFault*. O atributo *description* é utilizado para a descrição da técnica de tolerância a falhas do sistema.

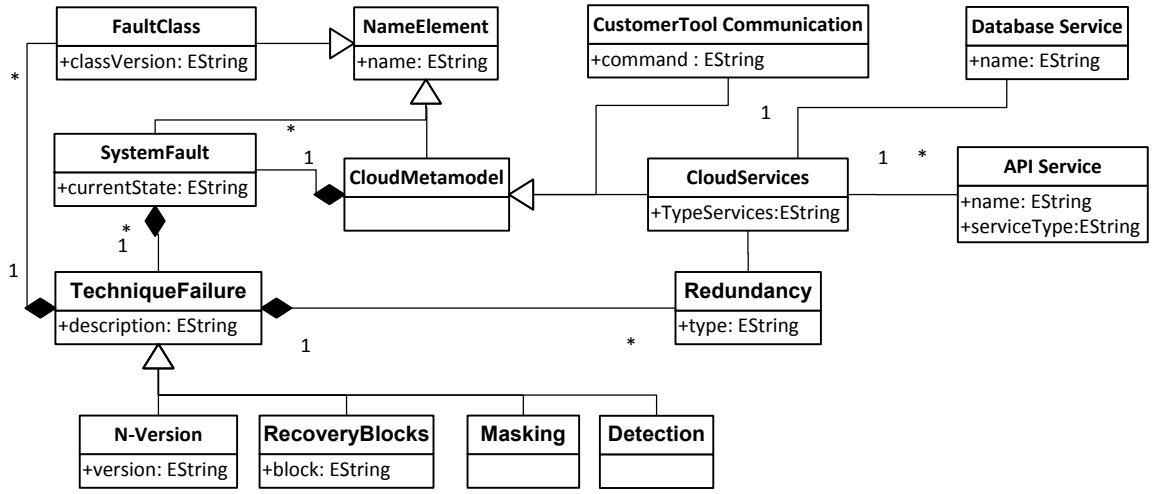
As técnicas utilizadas no metamodelo *FaultMetamodel* são a mais conhecidas no campo da tolerância a falhas. O metamodelo visa representar o uso das técnicas junto com a redundância de dados ou do próprio sistema. Como foi explicado na seção 2.3 a redundância aparece em quase todas as técnicas de falhas, por isso ela é representada como uma composição de *TechniqueFailure* na Figura 4.3.

Metamodelo de Tolerância a Falhas para a Nuvem: *CloudFault*

O metamodelo *CloudFault*, Figura 4.4, foi adaptado do trabalho de [5]. Nele há uma integração entre o metamodelo *FaultMetamodel*, Figura 4.3, com os elementos de

uma nuvem genérica.

Figura 4.4: Metamodelo de Tolerância a Falhas na Nuvem



Os elementos do metamodelo que representam a nuvem são:

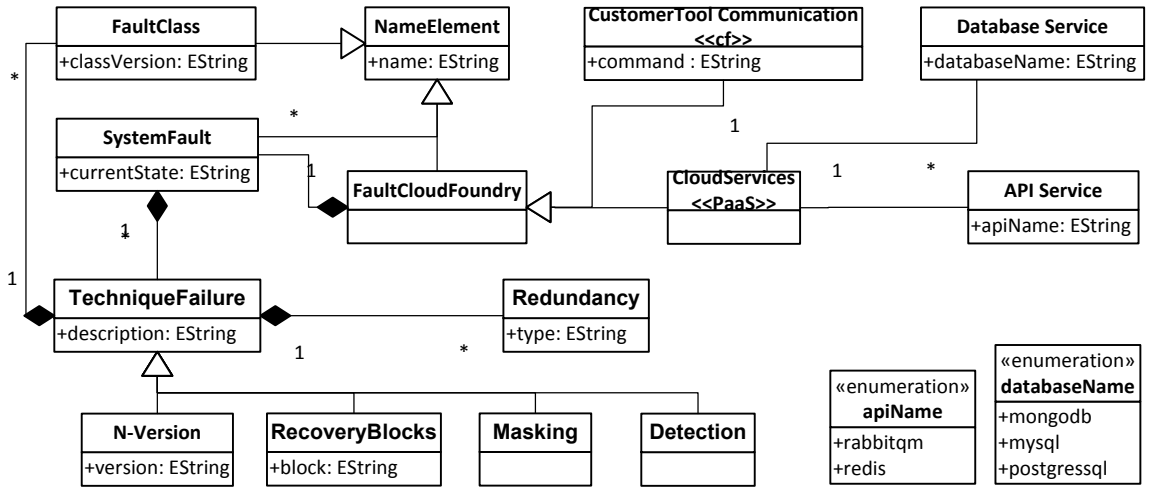
- *CloudMetamodel*: Elemento que representa a generalização dos elemento de uma nuvem;
- *CustomerToolCommunication*: Elemento que representa a comunicação entre a nuvem e o usuário;
- *CloudServices*: Elemento que representa o tipo de serviço prestado pela nuvem (PaaS, SaaS, IaaS);
- *Database Service*: Elemento que representa o serviço de banco de dados da nuvem, utilizado na redundância dos dados;
- *API Service*: Elemento que representa a API disponibilizada pela nuvem para que o desenvolvedor possa utilizar os serviços oferecidos pela mesma.

O metamodelo permite que a nuvem possa ter uma ou mais técnicas de tolerância a falhas. Os serviços das nuvem utilizam um tipo de redundância, dependendo do tipo de técnica empregada.

Metamodelo de Tolerância a Falhas para a Nuvem Específica: *FaultCloudFoundry*

A nuvem escolhida para o metamodelo específico foi a CloudFoundry por ser do tipo PaaS, desse modo, as técnicas de tolerância a falhas podem ser desenvolvidas junto com o sistema. O metamodelo de tolerância a falhas para nuvem específica é mostrado na Figura 4.5, ele foi baseado no metamodelo *CloudFoundry* do trabalho de [5] e adicionado os elementos de tolerância a falhas.

Figura 4.5: Metamodelo de Tolerância a Falhas na Nuvem CloudFoundry



O metamodelo possui alguns elementos específicos da nuvem *CloudFoundy*. Pode-se observar no metamodelo *FaultCloudFoundry*, que o elemento *CustomerToolCommunication* possui a ferramenta *CF* [1], específica para esta nuvem, e responsável pela comunicação entre o cliente e os seus serviços. O elemento *command* é o comando que o usuário pode fazer para realizar um tarefa na nuvem *CloudFoundry* como por exemplo, *login* ou *logout*.

O elemento *FaultCloudFoundry*, principal elemento do metamodelo, contém o *SystemFault* que representa o sistema que implementa as técnicas de tolerância a falhas *(TechniqueFailure)*. A classe *TechniqueFailure* possui o elemento *Redundancy*, com o tipo de redundância a ser utilizada. A classe *API Service* possui a *enumeration apiName*, que representa os nomes das APIs disponibilizadas pela nuvem. O elemento *DatabaseService* possui a *enumeration databaseName* com os nomes dos bancos de dados oferecidos pela nuvem.

4.4 Definições de transformação

As regras de transformação foram escritas na linguagem ATL, elas permitem a transformação semi-automática entre modelos. Neste trabalho, as seguintes definições foram utilizadas:

- **FaultMetamodel2CloudFault**: Definições de transformação utilizadas para geração do modelo de um plataforma da nuvem. O metamodelo *FaultMetamodel* é usado como entrada e como saída o *CloudFault*;
- **CloudFault2FaultCloudFoundry**: Regras de transformação entre um metamodelo de nuvem para um metamodelo de nuvem específica, no caso a nuvem *CloudFoundry*. Foi utilizado como entrada o metamodelo *CloudFault* e como saída o metamodelo *FaultCloudFoundry*.

Com a aplicação das regras de transformação, obtém-se um modelo no formato XMI (*XML Metadata Interchange*)[28].

A Listagem 4.1 mostra o fragmento da regra de transformação FaultMetamodel2CloudFault. Essa é a primeira transformação. Nela são definidos os componentes para uma nuvem, como o *TypeServices*.

Listagem 4.1: Fragmento da Regra de Transformação FaultMetamodel2CloudFault

```
1  @path CloudFault=/CloudFault/model/CloudFault.ecore
2  @path FaultMetamodel=/FaultMetamodel/model/FaultMetamodel.ecore
3
4  module faultMetamodel2cloudFault;
5  create OUT : CloudFault from IN : FaultMetamodel;
6
7  entrypoint rule CloudFault (){
8          to
9                  t: CloudFault!CloudServices (
10                         name <- 'Cloud Model',
11                         TypeServices <- 'PaaS'
12                 ),
13                 t1: CloudFault!APIServices (
```

```
14                          name <- 'redis'
15                  ),
16                  t2: CloudFault!DatabaseService(
17                          name <-          'mysql'
18                  )
19  }
20
21  rule SystmeFault2SystemFault {
22          from system: FaultMetamodel!SystemFault
23          to system2: CloudFault!SystemFault (
24                                  name <- system.name,
25                                  currentState <- system.currentState
26          )
27  }
28
29  rule FaultClass2FaultClass1 {
30          from class: FaultMetamodel!FaultClass
31          to class1: CloudFault!FaultClass (
32                          name <- class.name,
33                          classVersion <- class.classVersion
34                  )
35  }
36  ...
```

A Listagem 4.2 mostra o fragmento da regra de transformação CloudFault2-FaultCloudFoundry, essa é a segunda transformação modelo a modelo. Ela apresenta elemento específicos para a nuvem *CloudFoundry* como o *CustomerToolCommunication*.

Listagem 4.2: Fragmento da Regra de Transformação CloudFault2FaultCloudFoundry

```
1  @path CloudFault=/CloudFault/model/CloudFault.ecore
2  @path FaultFoundry=/FaultCloudFoundry/model/FaultCloudFoundry.ecore
3
4  module cloudMetamodel2foundryMetamodel;
5  create OUT : FaultFoundry from IN : CloudFault;
```

```
6
7  entrypoint rule CloudFault (){
8          to
9                  f: FaultFoundry!FaultCloudFoundry (
10                         name <- 'CloudFoundry'
11                 ),
12                 f1 : FaultFoundry!CustomerToolCommunication (
13                                 command <- 'vmc target api.clo
14                                 udfoundry.com'
15                 )
16 }
17
18 rule CustomerToolCommunication2CustomerToolCommunication1 {
19         from tools: CloudFault!CustomerToolCommunication
20         to tools1: FaultFoundry!CustomerToolCommunication(
21                 name <- tools.name
22         )
23 }
24
25 rule System2Systme1 {
26         from system: CloudFault!SystemFault
27         to system1: FaultFoundry!SystemFault (
28                 currentState <- system.currentState
29         )
30 }
31
32 rule FaultClass2FaultClass1 {
33         from class: CloudFault!FaultClass
34         to class1: FaultFoundry!FaultClass (
35                         name <- class.name,
36                         classVersion <- class.classVersion
37                 )
38 }
```

39 . . . . . .

As definições de transformação completas estão nos Anexos I.1 e I.2.

4.5 Síntese

Este capítulo apresentou a abordagem proposta para o desenvolvimento de um *framework* com suporte a Tolerância a Falhas para a Computação em Nuvem baseado em MDE.

Para o bom entendimento da abordagem, uma concepção básica foi apresentada. O *framework* para suportar Tolerãncia a Falhas em Computação em Nuvem foi explicado. O metamodelos propostos e as regras de transformação também foram apresentadas.

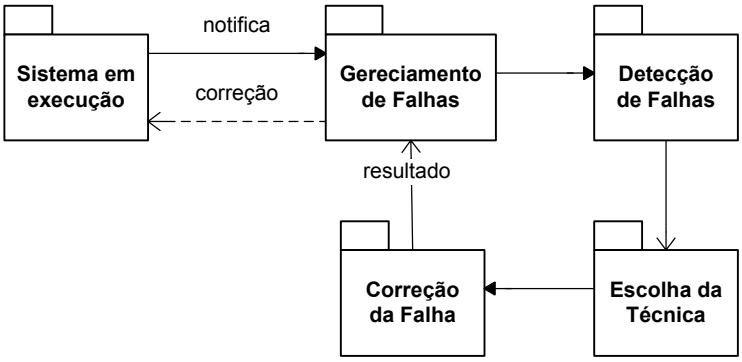
5 Prototipagem do *Framework* para Suportar Tolerância a Falhas

Esse capítulo descreve a implementação do *framework* de tolerância a falhas para a nuvem. Foi utilizado o EMF (*Eclipse Modeling Framework*) e o ambiente de desenvolvimento Eclipse. As regras de transformação foram feitas usando a linguagem ATL (*Atlas Transformation Language*).

5.1 Modelo do Protótipo

A Figura 5.1 mostra o protótipo do *framework*. O *framework* é formado por cinco módulos: o Sistema em Execução, o Gerenciamento de Falhas, o Detecção de Falhas, o Escolha da Técnica e a Correção da Falha.

Figura 5.1: Modelo do protótipo do *framework*



O módulo **Sistema em Execução** representa a aplicação rodando em perfeito estado, sem a presença de falhas. O módulo **Gerenciamento de Falhas** é o responsável por gerenciar todo o procedimento para a correção de uma falha, se uma falha acontecer, esse módulo chama o **Detecção de Falhas** que é o responsável por detectar a falha. O tipo de detecção pode ser escolhido pelo desenvolvedor como: erro no banco de dados ou uma grande demora na execução de um determinado processo.

Após a detecção da falha, o módulo **Escolha da Técnica** é chamado para a escolha da melhor técnica de acordo com o tipo de falha, nesse ponto uma das técnicas

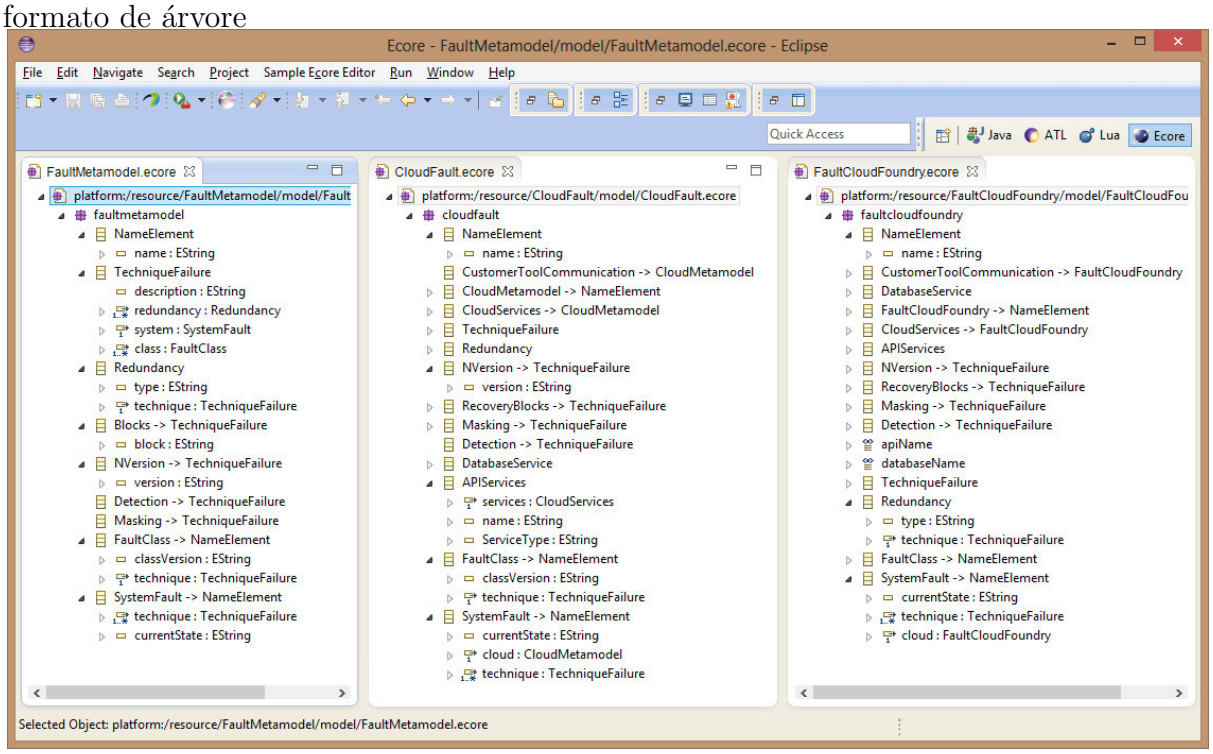
de falhas da Figura 4.3 pode ser utilizada como: *RecoveryBlocks*, *N-Version* ou *Masking*. Finalmente, o módulo **Correção de Falhas** é ativado, ele retorna uma mensagem para o **Gerenciador de Falhas** como o resultado da correção. Por fim, o módulo de gerenciamento faz a correção da falha no sistema em execução.

5.2 Plugin para Eclipse

Para o desenvolvimento do *plug-in* foi utilizada o EMF *(Eclipse Modeling Framework)* e também sua ferramenta o *GenModel* que permite a geração dos *plug-ins* para a criação e edição do modelos.

Para a criação do *plug-in*, primeiramente, foi necessário a construção dos metamodelos proposto na Figura 2.2, Figura 4.4 e Figura 4.5. Os metamodelos foram construídos utilizando o *Ecore metamodel*, a Figura 5.2 mostra os metamodelos exibidos em formato de árvore.

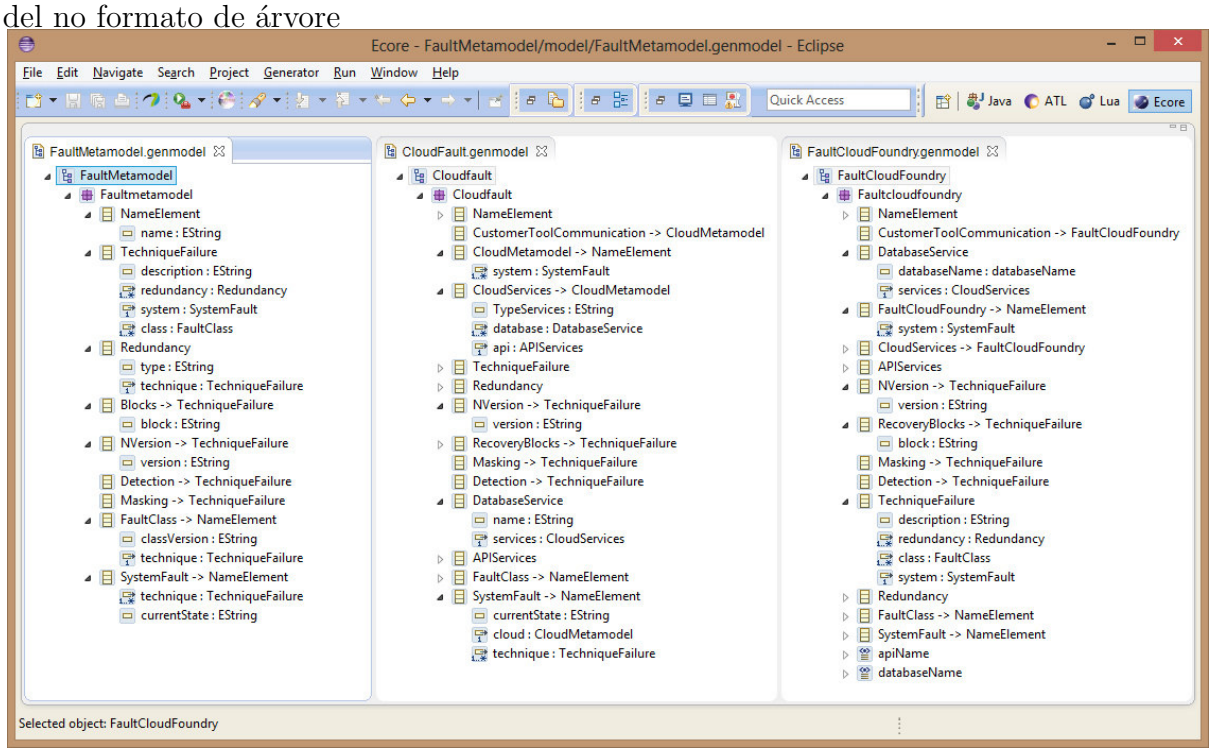
Figura 5.2: Metamodelos FaultMetamodel, CloudFault e FaultCloudFoudry em Ecore no formato de árvore



Após a criação dos metamodelos, a ferramenta GenModel do próprio EMF foi utilizada para a geração do *plug-in*. A Figura 5.3 mostra os metamodelos GenModel gerados a partir dos metamodelos *FaultTolerante Metamodel*, *CloudFault Metamodel* e

CloudFoundry Metamodel. Para a criação do GenModel foi necessário a validação dos metamodelos desejados e a importação do tipo de modelo, no caso o *Ecore model*. Com os metamodelos GenModel é possível gerar os *plug-ins* necessários para a criação e edição dos modelos.

Figura 5.3: Metamodelos FaultMetamodel, CloudFault e FaultCloudFoudry em GenModel no formato de árvore



A Figura 5.4 mostra a geração dos *plug-ins* para Eclipse. Neste trabalho, foram gerados todos os tipos de *plu-gins* oferecido pelo *framework* EMF Genmodel. Após a geração, clicando no menu *"Generate All"*, foram criados automaticamente pelo EMF três *plug-ings* com a extensão .edit, .editor e .tests.

Com os *plug-ins* gerados, a sua execução foi possível. A Figura 5.5 mostra o *plug-in* executando dentro do Eclipse, nele pode-se criar e editar modelos de acordo com o metamodelo escolhido, de forma gráfica no formato de árvore.

As transformações são executadas com as definições mostrada na seção 4.4 em que após a transformação do PIM, um modelo no formato XMI é gerado, como é explicado na próxima seção.

Figura 5.4: Geração dos *plu-gins* para Eclipse

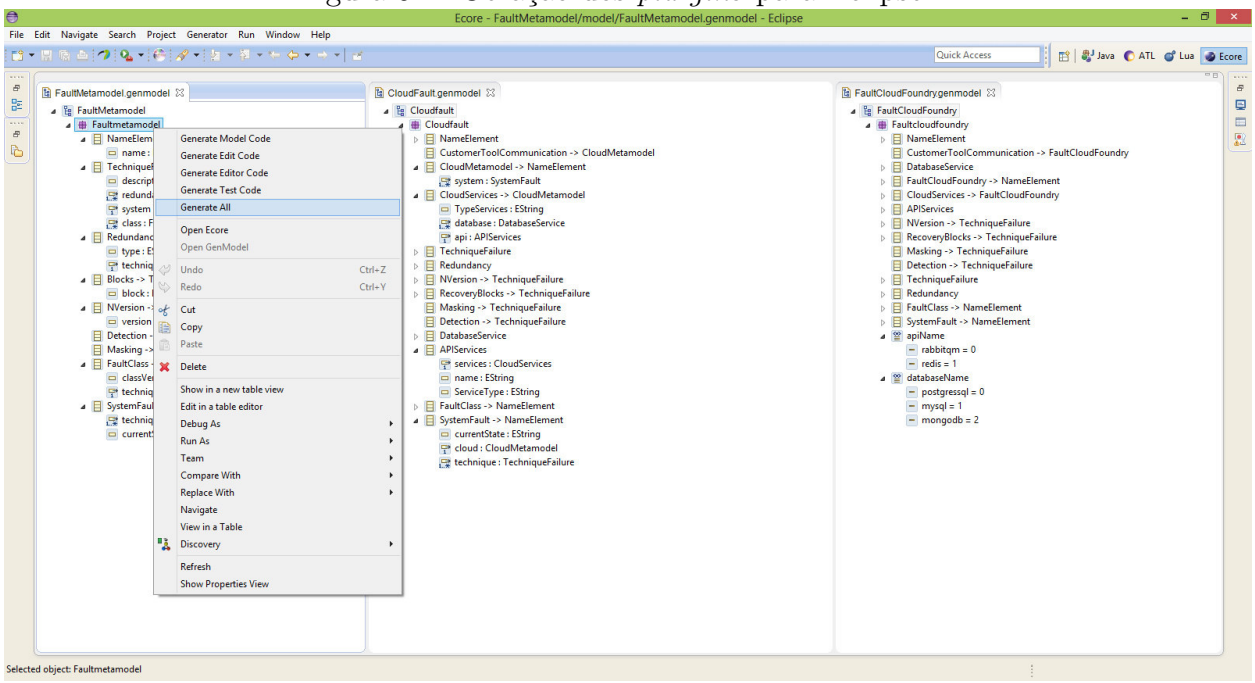
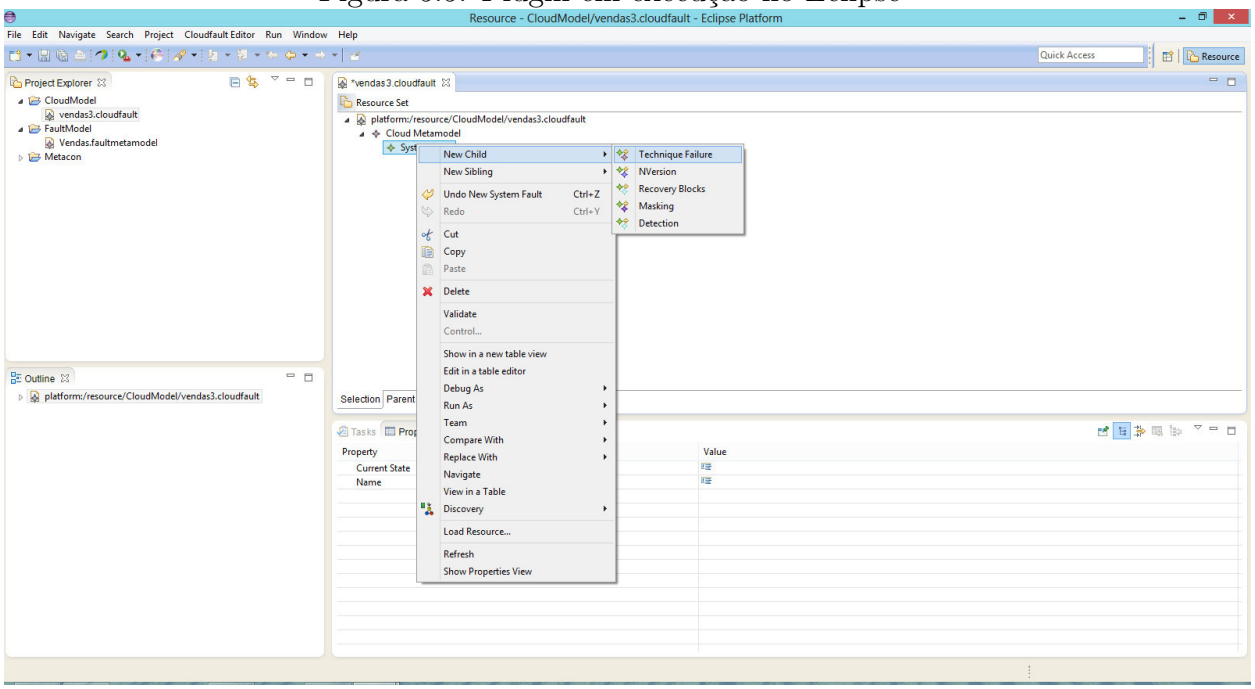


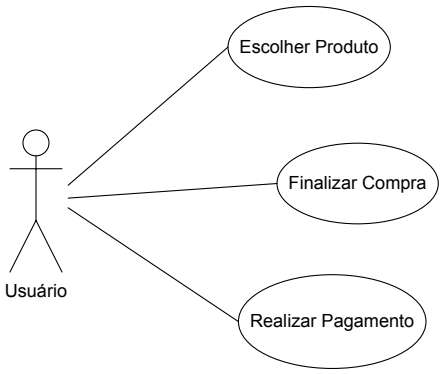
Figura 5.5: Plugin em execução no Eclipse



5.3 Exemplo Ilustrativo

Para um melhor entendimento do *framework* um exemplo ilustrativo é apresentado nesta seção. A aplicação desenvolvida consiste em um sistema de vendas pela *Web*. O usuário pode executar ações como: escolher produto, realizar pagamento e finalizar compra. A Figura 5.6 mostra o caso de uso do sistema.

Figura 5.6: Caso de Uso do sistema de Vendas *Web*



Quando o usuário requer alguma ação da Figura 5.6 e o sistema não responde em tempo satisfatório, tempo esse definido pelo desenvolvedor, ou retorna alguma mensagem de erro, o sistema de detecção de falhas do *framework* é ativado. Então a escolha da melhor técnica de falhas é realizada.

A Figura 5.7 representa o Diagrama de Classes da aplicação já modelado com as técnicas de tolerância a falhas. Esse modelo será transformado pelo *framework* para a obtenção do modelo de tolerância a falhas na nuvem. O modelo da Figura 5.7 foi criado dentro do *plug-in* no Eclipse.

A Figura 5.8 mostra a primeira transformação sendo executada, a **FaultMetamodel2CloudFault**, ela utiliza o metamodelo *FaultMetamodel.ecore* como entrada e o metamodelo *CloudFault.ecore* como o de saída. O modelo da Figura 5.7 foi o modelo fonte e após a execução das regras de transformação, o modelo *CloudFaultVendas.xmi* foi gerado. Esse processo é a transformação do modelo de Tolerância a Falhas para o modelo de Tolerância a Falhas Independente da plataforma de nuvem (PIM).

Após a geração do *CloudFaultVendas.xmi*, a segunda transformação é realizada, nela o metamodelo *CloudFault.ecore* é utilizado na entrada e o metamodelo de saída é o *FaultCloudFoundry.ecore* da nuvem *CloudFoudry*. Essa é a transformação **CloudFault2FaultCloudFoundry** que possui o modelo *CloudFaultVendas.xmi* como o fonte

Figura 5.7: Diagrama de Classes do sistema de Vendas *Web*

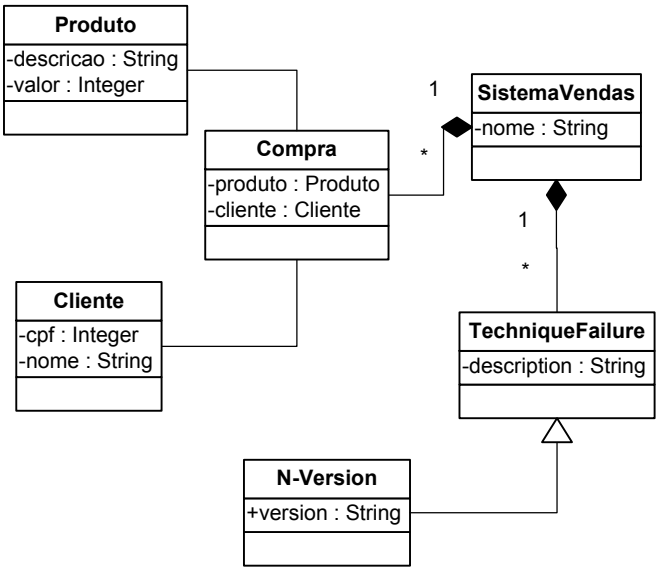
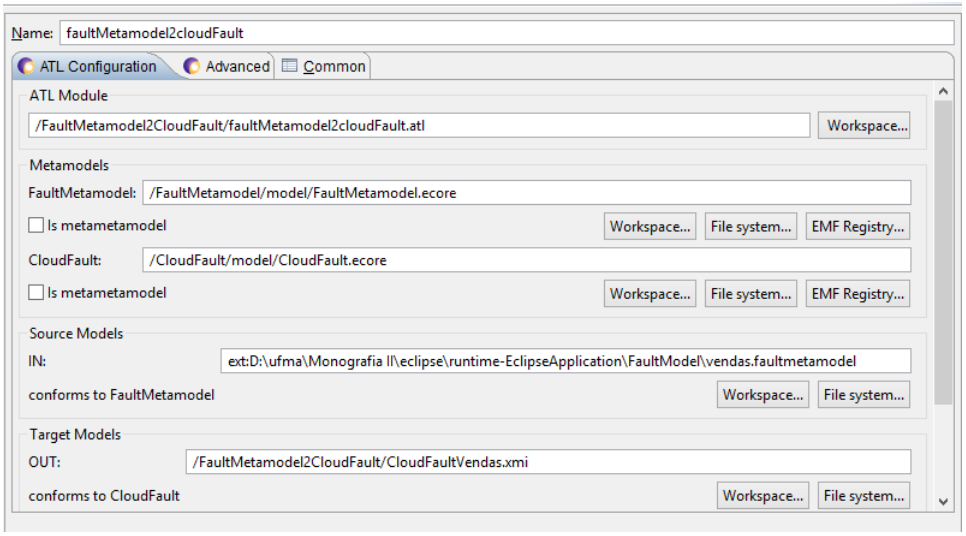


Figura 5.8: Processo de transformação FaultMematamodel2CloudFault



e o modelo *FoundryVendas.xmi* como alvo. Essa transformação pode ser visualizada na Figura 5.9.

Na Listagem 5.1 está o modelo *CloudFaultVendas.xmi* gerado pela transformação. Nele, os elementos TypeSerices indica o tipo de nuvem trabalhada e o elemento currentState é o responsável por manter a informação do estado do sistema, se apresenta algum tipo de erro ou não.

Listagem 5.1: Modelo PIM em XMI

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:cloudfault="http://cloudfault/1.0">
  <cloudfault:CloudServices name="Cloud Model"
  TypeServices="PaaS"/>
  <cloudfault:APIServices name="redis"/>
  <cloudfault:DatabaseService name="mysql"/>
  <cloudfault:SystemFault name="Sistema de Vendas"
  currentState="ok"/>
  <cloudfault:NVersion description="" version="2.0"/>
  <cloudfault:NVersion description="" version="3.0"/>
</xmi:XMI>
```

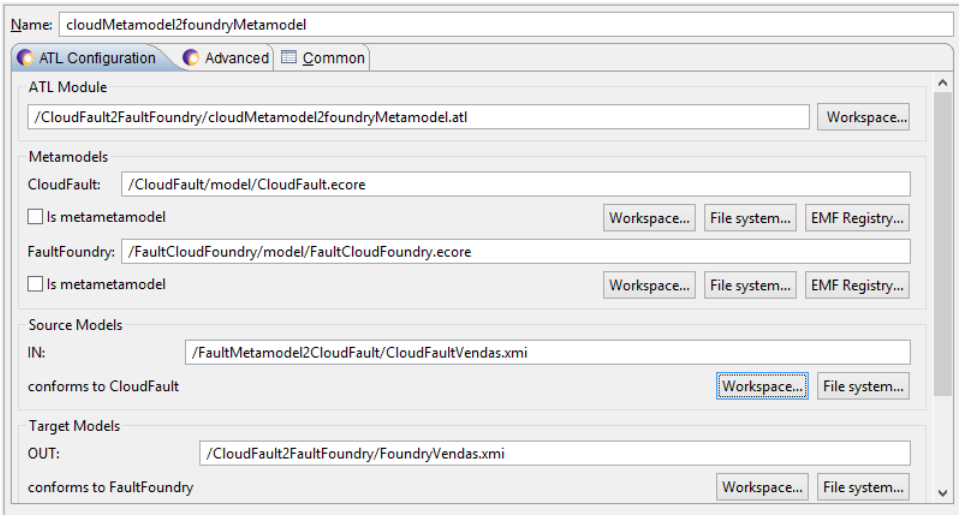
A transformação **CloudFault2FaultCloudFoundry** é a geração do modelo de Tolerância a Falhas Específico da Plataforma (PSM) a partir do PIM. O modelo *FoundryVendas.xmi* contém as característica específicas da nuvem *CloudFoundry*, pois está de acordo como o metamodelo *FaultCloudFoundry.ecore* definido na seção 4.3.

A Listagem 5.2 mostra o PSM gerado no formato XMI. Ele possui informações sobre sobre a plataforma específica *CloudFoundry* onde foram adicionadas durante o processo de transformação. O atributo *CustomerToolCommunication* é um elemento da nuvem *CloudFoundry* com comandos específicos para esta nuvem, a sua API também e representada pelo elemento APIService.

Os elementos de tolerância a falhas na Listagem 5.1 e na 5.2 são os mesmos, as diferenças são as especificações de cada nuvem.

Listagem 5.2: Modelo PSM em XMI

Figura 5.9: Processo de transformação CloudFault2FaultCloudFoundry




```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:faultcloudfoundry="http://faultcloudfoundry/1.0">
  <faultcloudfoundry:FaultCloudFoundry name="CloudFoundry"/>
  <faultcloudfoundry:CustomerToolCommunication
  command="vmc target api.cloudfoundry.com"/>
  <faultcloudfoundry:SystemFault currentState="ok"/>
  <faultcloudfoundry:CloudServices name="Cloud Model"
  TypeServices="PaaS"/>
  <faultcloudfoundry:APIServices apiName="redis"/>
  <faultcloudfoundry:NVersion description="" version="2.0"/>
  <faultcloudfoundry:NVersion description="" version="3.0"/>
  <faultcloudfoundry:DatabaseService databaseName="mysql"/>
</xmi:XMI>
```

O exemplo nesta seção demonstra a usabilidade do *plug-in*, mostrando que a partir da criação de uma aplicação que esteja de acordo como o *FaultMetamodel.ecore*, pode-se chegar em um modelo de uma plataforma específica de nuvem, através de um processo semi-automático de transformação.

5.4 Síntese

Neste capítulo, o modelo do protótipo do *framework* foi apresentado, nele foi possível visualizar como *framework* trabalha.

Um *plug-in* para Eclipse foi concebido no qual o EMF e o *GenModel* foi utilizado na sua construção. Os metamodelos foram implementados usando o *Ecore metamodel*. Um exemplo ilustrativo de um sistema de vendas *Web* foi implementado para mostra o funcionamento do *framework*.

6 Conclusões

Este capítulo discute os resultados deste trabalho. Os objetivos atingidos, as limitações verificadas no decorrer da pesquisa, os trabalhos futuros e as recomendações são apresentados.

6.1 Objetivos Atingidos

Este trabalho de conclusão apresentou uma abordagem baseada em Engenharia Dirigida por Modelos para suportar a Computação em Nuvem com foco em Tolerância a Falhas. Além da abordagem, estudos e pesquisas na áreas da MDE, Tolerância a Falhas e Computação em Nuvem foram feitos visando um maior aprendizado e buscando novas tecnologias para este trabalho.

Neste trabalho três metamodelos foram desenvolvidos:

- Um metamodelo de Tolerância a Falhas sem as características da Computação em Nuvem (*FaultMetamodel.ecore*);
- Um metamodelo de Tolerância a Falhas Independente da plataforma de nuvem (*FaultCloud.ecore*);
- Um metamodelo específico da plataforma de nuvem (*CloudFoundry.ecore*).

Definições de transformação em ATL foram construídas, possibilitando o processo de transformação semi-automáticos entre os modelos. As regras de transformação: *FaultMetamodel2CloudFault*, *ClouFault2FaultCloudFoundry* foram desenvolvidas.

Um *framework* baseado em MDE para suportar a Computação em Nuvem com foco em Tolerância a Falhas foi proposto. Com o *framework*, o usuário pode se concentrar mais nas soluções dos problemas da aplicação do que nos detalhes técnicos de um plataforma de nuvem.

Um *plug-in* para o Eclipse foi concebido utilizando o *EMF*, a ferramenta *GenModel* e o *Ecore metamodel*.

6.2 Limitações

O trabalho alcançou alguns dos principais objetivos, mas existem limitações que podem ser melhoradas, tais como:

- O *framework* foi direcionado apenas em uma plataforma de nuvem, faltando a sua utilização em outras plataforma para verificar melhor a sua interoperabilidade, encontrar erros e adicionar melhorias;
- As definições de transformação foram feitas em ATL, mas podem ser utilizadas também a MOF/QVT;
- Poucos testes foram realizados no *framework*, diminuindo assim a verificação de sua eficácia em diferentes sistemas de nuvem.

6.3 Trabalhos Futuros

Os trabalho futuros e aprimoramentos são apresentados nesta seção, tais como listados a seguir:

- Construção de um ambiente gráfico para a criação e edição dos modelos utilizados no *plug-in*;
- Criação de *wizards* para importação dos modelos no *plug-in*;
- Desenvolver metamodelos para outras plataformas de nuvem. O *framework* trabalho apenas como o metamodelo da nuvem *CloudFoundry*, mas pode ser facilmente estendido para outra plataformas de Computação em Nuvem;
- Testar outras técnicas de Tolerância a Falhas além da *n-version* utilizada no exemplo ilustrativo;
- Criação de definições de transformação em MOF/QVT.
- Melhorar as regras de transformação do *CloudFoundry* para código;
- Utilização de ferramentas para facilitar a geração das regras de transformação como MT4MDE e SAMT4MDE [21][20].

- Analisar o custo beneficio do *framework* e em quais situações de falhas a sua utilização é viável.

Referências Bibliográficas

[1] Command line interface (cli), Novembro 2014. Disponivel em: http://docs.cloudfoundry.org/devguide/installcf/ Acesso em Novembro de 2014.

[2] M. Abdalla, M. Hassan, J. Jaafar, and L. Rahim. Enhanced approach for developing web applications using model driven architecture. *International Conference on Research and Innovation in Information Systems*, pages 145 – 150, Novembro 2013.

[3] Y. Amanatullah, C. Lim, H. Ipung, and A. Juliandri. Toward cloud computing reference architecture: Cloud service management perspective. *ICT for Smart Society (ICISS), 2013 International Conference on*, pages 1 – 4, Julho 2013.

[4] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE*, 1985.

[5] J. Bassani. Uma abordagem baseada em engenharia dirigida por modelos para suportar o teste de sistemas de software na plataforma de computacao em nuvem. ufma, 2012.

[6] M. Clifton. What is a framework?, Novembro 2003. Disponivel em: http://www.codeproject.com/KB/ architecture/WhatIsAFramework.aspx. Acesso em Setembro de 2011.

[7] D. Doering, C. Pereira, P. Denes, and J. Joseph. A model driven engineering approach based on aspects for high speed scientific x-rays cameras. IEEE, Junho 2013.

[8] R. Ferraz. Domain specific languages: Introdução, 2008. Disponivel em:http://logbr.reflectivesurface.com/page/7/?s=cFevereiro de 2012.

[9] E. Foundation. Eclipse modeling framework (emf), 01 2015. Disponivel em: http://eclipse.org/modeling/emf/.

[10] J. Greenfield and K. Short. Software factories: Assembling applications with patterns, models, frameworks, and tools. *https://msdn.microsoft.com/en-us/library/ms954811.aspx*, November 2004.

[11] G. Guedes. *UML 2 Uma Abordagem Pratica*. 2011.

[12] L. . INRIA. *ATL User Manual*, version 0.7 edition, 2006.

[13] Y. Jadeja and K. Modi. Cloud computing - concepts, architecture and challenges. *International Conference on Computing, Electronics and Electrical Technologies [ICCEET]*, pages 1 – 4, 2012.

[14] R. Jhawar, V. Piuri, and M. Santambrogio. A comprehensive conceptual system-level approach to fault tolerance in cloud computing. *Systems Conference (SysCon), 2012 IEEE International*, pages 1 – 5, Marco 2012.

[15] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[16] S. Lakshmi. Fault tolerance in cloud computing. *International Journal of Engineering Sciences Research-IJESR*, 04(01):1 – 4, 2013.

[17] G. Lewis. Role of standards in cloud-computing interoperability. *System Sciences (HICSS), 2013 46th Hawaii International Conference on*, Janeiro 2013.

[18] D. Lopes. Linguagens de transformacao de modelos, 2006.

[19] D. Lopes. Introducao a engenharia dirigida por modelos. *I Escola Regional de Computacao Ceara Maranhao Piaui (ERCEMAPI)*, 2007.

[20] D. Lopes, S. Hammoudi, and Z. Abdelouahab. Schema matching in the context of model driven engineering: From theory to practice. *Proceedings of the International Conference on Systems, Computing Sciences and Software Engineering (SCSS 2005)*, 2005.

[21] D. Lopes, S. Hammoudi, J. Bezivin, and F. Jouault. *Mapping Specification in MDA: From Theory to Practice*. Janeiro 2005.

[22] Loutas, Hellas, Thessaloniki, G. Kamateri, Bosi, and Tarabanis. Cloud computing interoperability: The state of play. *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, 2011.

[23] E. Nogueira, V. da Silva, D. Lucredio, and R. Fortes. Towards a model-driven approach for promoting cloud paas portability. *XXXIX Latin American Computing Conference (CLEI)*, pages 1 – 9, Outubro 2013.

[24] J. Oliveira, D. Lopes, Z. Abdelouahab, D. Claro, and S. Hammoudi. Model driven testing for cloud computing. *Springer Verlag - Lecture Notes in Electrical Engineering - Innovations and Advances in Computer, Information, Systems Sciences, and Engineering*, 313(297-304), 2014.

[25] OMG. Mda overview. 2001.

[26] OMG. Mda guide. 2003.

[27] OMG. Meta object facility (mof) 2.0 query/view/transformation specification. 2011.

[28] OMG. Xml metadata interchange (xmi) specification, Abril 2014.

[29] OMG. Unified modeling language, 2015. Disponivel em: http://www.uml.org/.

[30] M. Regio and J. Greenfield. A software factory approach to hl7 version 3 solutions. 2005.

[31] D. C. Schmidt. Model-driven engineering. *IEEE Computer*, February 2009.

[32] Sommerville. *Engenharia de Software*. Pearson, 8 edition, 2007.

[33] B. Sosinsky. *Cloud Computing Bible*. Wiley Publishing, 2011.

[34] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF Eclipse Modeling Framework*. Addison-Wesley, 2008.

[35] Troyer and Leune. Wsdm: A user centered design method for web sites. *Computer Networks and ISDN Systems*, 1998.

[36] T. Weber. Um roteiro para exploração dos conceitos basicos de tolerância a falhas.

[37] A. Ziani, B. Hamid, and J.-M. Bruel. A model-driven engineering framework for fault tolerance in dependable embedded systems design. *38th Euromicro Conference on Software Engineering and Advanced Applications*, pages 1–4, 2012.

I Anexo - Regras de Transformação

I.1 Anexo A - Regras de Transfomação de *FaultMetamodel* para *CloudFault*

```
1  @path CloudFault=/CloudFault/model/CloudFault.ecore
2  @path FaultMetamodel=/FaultMetamodel/model/FaultMetamodel.ecore
3
4  module faultMetamodel2cloudFault;
5  create OUT : CloudFault from IN : FaultMetamodel;
6
7  entrypoint rule CloudFault (){
8          to
9                  t: CloudFault!CloudServices (
10                         name <- 'Cloud Model',
11                         TypeServices <- 'PaaS'
12                 ),
13                 t1: CloudFault!APIServices (
14                         name <- 'redis'
15                 ),
16                 t2: CloudFault!DatabaseService(
17                         name <-          'mysql'
18                 )
19 }
20
21 rule SystmeFault2SystemFault {
22         from system: FaultMetamodel!SystemFault
23         to system2: CloudFault!SystemFault (
24                                 name <- system.name,
25                                 currentState <- system.currentState
```

```
26          )
27  }
28
29  rule FaultClass2FaultClass1 {
30          from class: FaultMetamodel!FaultClass
31          to class1: CloudFault!FaultClass (
32                          name <- class.name,
33                          classVersion <- class.classVersion
34                  )
35  }
36
37  rule Redundancy2Redundancy1 {
38          from redundancy: FaultMetamodel!Redundancy
39          to redundancy1: CloudFault!Redundancy (
40                          type <- redundancy.type,
41                          technique <- redundancy.technique
42                  )
43  }
44
45  rule RecoveryBlocks2RecoveryBlocks1 {
46          from block: FaultMetamodel!RecoveryBlocks
47          to block1: CloudFault!RecoveryBlocks (
48                  description <- block.description
49          )
50  }
51
52  rule NVersion2NVersion1 {
53          from nversion: FaultMetamodel!NVersion
54          to nversion1: CloudFault!NVersion (
55                  description <- nversion.description,
56                  version <- nversion.version
57          )
58  }
```

```
59
60 rule Detection2Detection1 {
61         from detection: FaultMetamodel!Detection
62         to detection1: CloudFault!Detection (
63                 name <- detection.name
64         )
65 }
66
67 rule Masking2Masking1 {
68         from masking: FaultMetamodel!Masking
69         to masking1: CloudFault!Masking (
70                 name <- masking.name
71         )
72 }
```

I.2 Anexo B - Regras de Transfomação de *CloudFault* para *FaultCloudFoundry*

```
1  @path CloudFault=/CloudFault/model/CloudFault.ecore
2  @path FaultFoundry=/FaultCloudFoundry/model/FaultCloudFoundry.ecore
3
4  module cloudMetamodel2foundryMetamodel;
5  create OUT : FaultFoundry from IN : CloudFault;
6
7  entrypoint rule CloudFault (){
8          to
9                  f: FaultFoundry!FaultCloudFoundry (
10                         name <- 'CloudFoundry'
11                 ),
12                 f1 : FaultFoundry!CustomerToolCommunication (
13                                 command <- 'vmc target api.cloud
14                                 foundry.com'
15                 )
```

```
16 }
17
18 rule CustomerToolCommunication2CustomerToolCommunication1 {
19         from tools: CloudFault!CustomerToolCommunication
20         to tools1: FaultFoundry!CustomerToolCommunication(
21                 name <- tools.name
22         )
23 }
24
25 rule System2Systme1 {
26         from system: CloudFault!SystemFault
27         to system1: FaultFoundry!SystemFault (
28                 currentState <- system.currentState
29         )
30 }
31
32 rule FaultClass2FaultClass1 {
33         from class: CloudFault!FaultClass
34         to class1: FaultFoundry!FaultClass (
35                         name <- class.name,
36                         classVersion <- class.classVersion
37                 )
38 }
39
40 rule CloudServices2CloudServices1{
41         from cloudservices: CloudFault!CloudServices
42         to cloudservices1: FaultFoundry!CloudServices(
43                 name <- cloudservices.name,
44                 TypeServices <- cloudservices.TypeServices
45         )
46 }
47
48
```

```
49 rule APIServices2APIServices1{
50         from apiservices: CloudFault!APIServices
51         to apiservices1: FaultFoundry!APIServices(
52                 apiName <- apiservices.name,
53                 services <- apiservices.services
54         )
55 }
56
57 rule Redundancy2Redundancy1 {
58         from redundancy: CloudFault!Redundancy
59         to redundancy1: FaultFoundry!Redundancy (
60                         name <- redundancy.name,
61                         type <- redundancy.type
62                 )
63 }
64
65 rule Block2Block1 {
66         from block: CloudFault!RecoveryBlocks
67         to block1: FaultFoundry!RecoveryBlocks (
68                 name <- block.name,
69                 description <- block.description
70         )
71 }
72
73 rule NVersion2NVersion1 {
74         from nversion: CloudFault!NVersion
75         to nversion1: FaultFoundry!NVersion (
76                 description <- nversion.description,
77                 version <- nversion.version
78         )
79 }
80
81 rule Detection2Detection1 {
```

```
82          from detection: CloudFault!Detection
83          to detection1: FaultFoundry!Detection (
84                  name <- detection.name
85          )
86  }
87
88  rule Masking2Masking1 {
89          from masking: CloudFault!Masking
90          to masking1: FaultFoundry!Masking (
91                  name <- masking.name
92          )
93  }
94
95  rule DatabaseService2DatabaseService1 {
96          from database: CloudFault!DatabaseService
97          to database1: FaultFoundry!DatabaseService (
98                  databaseName <- database.name,
99                  services <- database.services
100         )
```