PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Universidade Federal do Rio de Janeiro

Marcos Vinícius Furriel Amorim Dias

UMA ABORDAGEM SISTEMÁTICA PARA GERAÇÃO DE CASOS DE TESTE ABSTRATOS UTILIZANDO MODELOS DE CLASSES UML ANOTADOS COM RESTRIÇÕES OCL

DISSERTAÇÃO DE MESTRADO

Instituto de Matemática

Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
INSTITUTO TÉRCIO PACITTI DE APLICAÇÕES E PESQUISAS COMPUTACIONAIS
PROGRAMA DE PÓS-GRADUAÇÃOEM INFORMÁTICA

MARCOS VINICIUS FURRIEL AMORIM DIAS

Uma Abordagem Sistemática para Geração de Casos de Teste Abstratos utilizando Modelo de Classes UML Anotados com Restrições OCL

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática, Instituto de Matemática e Instituto Tércio Pacitti, Universidade Federal do Rio de Janeiro, como requisito parcial à obtenção do título de Mestre em Informática.

Orientador: Prof. Eber Assis Schmitz, Ph.D

Rio de Janeiro
2017

CIP - Catalogação na Publicação

D541a

Dias, Marcos Vinícius Furriel Amorim
Uma Abordagem Sistemática para Geração de Casos de Teste Abstratos utilizando Modelo de Classes UML Anotados com Restrições OCL / Marcos Vinícius Furriel Amorim Dias. -- Rio de Janeiro, 2017.
112 f.

Orientador: Eber Assis Schmitz.
Dissertação (mestrado) - Universidade Federal do Rio de Janeiro, Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais, Programa de Pós-Graduação em informática, 2017.

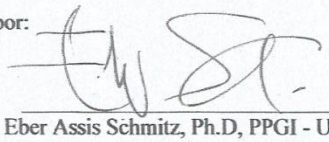
1. Teste de Software. 2. Teste Baseado em Modelos. 3. Sistemas de Informação. I. Schmitz, Eber Assis, orient. II. Título.

Uma Abordagem Sistemática para Geração de Casos de Teste Abstratos utilizando Modelos de Classe UML Anotados com Restrições OCL

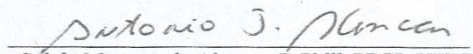
Marcos Vinicius Furriel Amorim Dias

Dissertação de Mestrado submetida ao Programa de Pós-graduação em Informática do Instituto de Matemática e do Instituo Tércio Pacitti da Universidade Federal do Rio de Janeiro - UFRJ, como parte dos requisitos necessários à obtenção do título de Mestre em Informática.

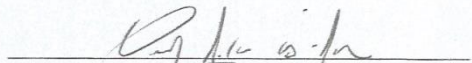
Aprovada em 26 de Junho de 2017 por:



Prof. Eber Assis Schmitz, Ph.D, PPGI - UFRJ (Presidente)



Prof. Antonio Juarez Sylvio Menezes de Alencar, D.Phill, PPGI- UFRJ



Prof. Denis Silva da Silveira, DSc, UFPE

Aos meus familiares

Agradecimentos

Agradeço à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pela bolsa de estudos de Mestrado.

Aos meus pais Marcos e Sheila, e demais familiares, pelo apoio fornecido em minha vida até aqui.

A minha companheira Roberta, que me apoia de maneira incondicional nessa minha caminhada acadêmica.

Ao meu orientador Prof. Eber Assis Schmitz por toda a atenção, paciência, compreensão e dedicação fornecida para que eu pudesse aprender tudo aquilo me foi transmitido nesse período de estudos.

Aos professores da Universidade do Grande Rio (Unigranrio), em especial ao Prof. Thiago Silva de Souza, pelo apoio dado não só na graduação como também durante o período da pós-graduação.

Aos professores do Programa de Pós Graduação em Informática da UFRJ (PPGI), em especial à Prof.ª Mônica Ferreira da Silva, pelos ensinamentos e conselhos que me foram dados. Aos alunos da professora Mônica que me deram suporte relacionado à escrita da dissertação.

Aos professores Denis Silva da Silveira e Antônio Juarez Alencar que gentilmente aceitaram o convite para participarem da Banca para defesa.

Ao pessoal da Secretaria do PPGI, por toda a ajuda e esclarecimento prestados durante todo esse período de aprendizado.

E por fim, agradeço a todos aqueles que torceram por mim nessa jornada de dedicação e empenho, e que certamente contribuíram para a conclusão deste trabalho.

“Para realizar grandes conquistas, devemos não apenas agir, mas também sonhar, não apenas planejar, mas também acreditar"
(Anatole France)

Resumo

DIAS, Marcos Vinicius Furriel Amorim. **Uma Abordagem Sistemática para Geração de Casos de Teste Abstratos utilizando Modelo de Classes UML Anotados com Restrições OCL. 2017**. 112f. Dissertação (Mestrado em Informática) – Instituto de Matemática, Instituto Tércio Pacciti, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2017.

Esta dissertação descreve uma abordagem sistemática para geração de casos de teste independentes de plataforma utilizando a ferramenta USE. A proposta segue os princípios encontrados nas estratégias *lightweight* de validação de modelos e tem como público alvo os analistas de requisitos especializados na especificação de modelos conceituais utilizando UML, OCL e ASSL. A principal contribuição desta dissertação está relacionada à criação de métodos baseados na hipótese do escopo reduzido utilizando o gerador de *snapshots* do USE através da linguagem ASSL juntamente com os critérios de seleção conhecidos em teste de *software*, como particionamento em classes de equivalência e análise do valor-limite. O objetivo é possibilitar a geração dos casos de teste e oráculos de teste através da verificação e validação das instâncias de um modelo conceitual corretamente especificado por meio do diagrama de classes da UML e de restrições especificadas em OCL. Para isso são adotados critérios de cobertura baseados em multiplicidade, atributos e generalização. Um quase-experimento e um exemplo de utilização foram utilizados para demonstrar a viabilidade da abordagem. A partir dos resultados obtidos foi possível obter evidências de que a proposta do trabalho é viável e eficaz para cobertura do teste.

Palavras-chave: Teste de *Software,* UML*,* OCL, ASSL, Teste Baseado em Modelos.

Abstract

DIAS, Marcos Vinicius Furriel Amorim. **Uma Abordagem Sistemática para Geração de Casos de Teste Abstratos utilizando Modelo de Classes UML Anotados com Restrições OCL. 2017**. 112f. Dissertação (Mestrado em Informática) – Instituto de Matemática, Instituto Tércio Pacciti, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2017.

This dissertation describes a systematic approach for generating platform-independent test cases using the USE tool. The proposal follows the principles found in lightweight model validation strategies and targets the requirements analysts specialized in constructing conceptual models using UML, OCL and ASSL. The main contribution of this dissertation is related to the creation of methods based on the small scope hypothesis and the USE snapshot generator language called ASSL together with the known selection criteria in Software Testing, such as equivalence class partitioning and boundary value analysis. The objective is to enable the generation of test cases and oracles through the verification and validation of the instances of the conceptual model expressed through the UML Class Diagram annotated with OCL constraints. Coverage criteria based on multiplicity, attributes and generalization are adopted. A quasi-experiment and an example of use were used to demonstrate the feasibility of approaches. From the obtained results it was possible to obtain evidences that the work proposal is feasible and effective to cover the test.

Keywords: *Software* Testing, UML, OCL, ASSL, Model-Based Testing

Lista de Figuras

Lista de Tabelas

Lista de Siglas

ASSL	*A Snapshot Sequence Language*
CRUD	*Create, Retrieve, Update and Delete*
MDA	*Model-Driven Architecture*
MDSE	*Model-Driven Software Engineering*
MOF	*Meta-Object Facility*
OCL	*Object Constraint Language*
OMG	*Object Management Language*
SOIL	*Simple OCL-Based Imperative Programming*
SST	Sistema Sob Teste
TBM	*Teste Baseado em Modelos*
UML	*Unified Modeling Language*
USE	*UML-Based Specification Environment*
V&V	Verificação e Validação
XML	*eXtensible Markup Language*

Sumário

1 Introdução

Este capítulo está dividido em cinco seções. A seção 1.1 apresenta a contextualização do tema abordado. A seção 1.2 apresenta a motivação para realização deste trabalho. A seção 1.3 descreve o problema a ser tratado. A seção 1.4 os objetivos e o escopo deste trabalho. Na seção 1.5 é apresentada a metodologia empregada. Por fim, a seção 1.6 descreve a estrutura segundo a qual este trabalho está organizado.

1.1 Contextualização

O teste de *software* é uma parte importante relacionada às atividades de Verificação e Validação (V&V) que visam garantir a qualidade do produto sendo desenvolvido. Os processos de V&V avaliam se o sistema em desenvolvimento atende às especificações dos requisitos (funcionais e não-funcionais) e à satisfação do cliente (PRESSMAN, 2011; SOMMERVILLE, 2011).

A busca por qualidade acaba por gerar esforços que elevam o custo e o tempo para desenvolvimento do *software,* principalmente quando apresentam alto nível de complexidade (SILVA-DE-SOUZA, 2012; SOUSA, 2009). Esses esforços que são gastos a partir do processo de teste estão divididos entre as fases do ciclo de vida do teste. Essas fases estão relacionadas ao planejamento, preparação, projeto, execução e entrega dos testes (IEEE, 2008). De acordo com Naik e Tripathy (2008), a fase de projeto de teste é uma etapa do teste de *software* considerada crítica, pois é nela onde os requisitos do sistema são compreendidos, os recursos do sistema a serem testados são bem identificados e os objetivos e comportamento dos casos de teste são definidos. Esses objetivos são definidos a partir de diferentes meios, como por exemplo, a especificação dos requisitos e um modelo conceitual do domínio, de maneira que um ou mais casos de teste sejam concebidos para cada requisito identificado (NAIK e TRIPATHY, 2008).

Devido ao aumento da complexidade dos sistemas da informação, diversas abordagens baseadas em modelos surgiram. Tais abordagens têm como objetivo gerar sistemas a partir de modelos conceituais que representam o domínio com alto nível de abstração (BRAMBILLA, CABOT e WIMMER, 2012). Nesse caminho surgiu também o Teste

Baseado em Modelos (TBM), cujo propósito é gerar testes para sistemas a partir de suas especificações.

Os modelos conceituais do sistema, quando bem especificados, representam a estrutura e o comportamento completo da aplicação. Desse modo, é necessário que o processo de modelagem do *software* também experimente técnicas que visem diminuir as incertezas relacionadas ao processo de modelagem dos requisitos. Dentro desse contexto, surgiram algumas técnicas para V&V das especificações do sistema, que permitem ao analista avaliar se o modelo realmente está de acordo com os requisitos, como é o caso das técnicas conhecidas como *heavyweight* e *lightweight* (JACKSON, 1996; JACKSON, 2006).

As técnicas *heavyweight* utilizam provas formais para verificação das inconsistências em modelos, tornando a sua utilização difícil até mesmo para analistas experientes, pois é necessário alto conhecimento sobre métodos formais e lógica matemática. Em contrapartida, as técnicas *lightweight* surgiram com o intuito de tornar essa verificação mais simples e menos custosa, possibilitando ao analista uma maior facilidade para encontrar inconsistências em seus modelos (JACKSON, 1996; JACKSON, 2006; ARAÚJO, 2010). Por fim, modelos conceituais livres de inconsistências que representam a estrutura e o comportamento de um sistema sob teste permitem que testes possam ser gerados a fim de que a qualidade do produto desenvolvido seja garantida.

1.2 Motivação

A dificuldade de garantir qualidade ao *software* cresce consideravelmente à medida que a complexidade dos sistemas aumenta (SILVA-DE-SOUZA, 2012; SOUA, 2009). As abordagens de desenvolvimento orientadas a modelos (do inglês, *Model Driven Software Engeenering* - MDSE) permitem concentrar-se na essência do sistema sem que haja qualquer tipo de interferência de tecnologia (JACKSON, 2006; BRAMBILLA, CABOT e WIMMER, 2012).

O mesmo tipo de abordagem pode ser aplicado à criação de casos de teste. Ao escolher uma abordagem orientada a modelos uma parte desta complexidade é abstraída do processo de criação de casos de teste já que o analista passa criar seus modelos de teste independentemente da plataforma utilizada. Em particular, a criação de casos de teste a partir de modelos conceituais, que compreendem as entidades do domínio, seus relacionamentos e as restrições de integridade, diminui o tempo e os recursos gastos durante o desenvolvimento. Além disso, é uma garantia de que o produto vai atender

verdadeiramente aos requisitos especificados, sem passar pela interferência da interpretação do analista de testes (UTTING e LEGEARD, 2007; SOUSA, 2009).

1.3 Problema

A criação manual de Casos de Teste por vezes é muito trabalhosa, sendo uma tarefa que compromete tempo e recursos por parte dos analistas de teste (SOUSA, 2009). Em segundo lugar, o trabalho de criação manual dos casos de teste pode ser considerado uma tarefa entediante, afetando a forma como os testes são planejados e construídos (ISMAIL, 2007). Como consequência, o analista de teste pode não alcançar uma cobertura satisfatória das situações a serem testadas, prejudicando a qualidade esperada para o *software* (SILVA-DE-SOUZA, 2012).

Os maiores esforços durante o ciclo de desenvolvimento de um *software* se concentram na fase de teste (SOUSA, 2009), que chegam a atingir entre 30% e 60% dos custos de desenvolvimento de *software* (UTTING e LEGEARD, 2007; BAO-LIN *et al*., 2007) e (DIANXIANG et al, 2015). Nessa fase é notado um grande consumo de tempo e recursos por parte das produtoras de *software,* aumentando o custo necessário para produção do sistema. Utting e Legeard (2007) estimam que o tempo necessário para a especificação do teste compreende mais de 20% do tempo que é gasto com a execução deles.

As estratégias *lightweight* de validação do modelo conceitual e a utilização da TBM representam uma possibilidade de aumentar a qualidade do *software* com baixo custo de investimento. Se essas estratégias forem aplicadas para geração de casos de teste, podem possibilitar que os esforços gastos na fase de especificação do teste sejam diminuídos. No entanto, a utilização desse tipo de abordagem pode ocasionar um crescimento exponencial do número de casos de teste gerados.

1.4 Objetivos

O objetivo geral deste trabalho é apresentar uma abordagem para a geração de casos de teste abstratos a partir de modelos de classes da UML (do inglês, *Unified Modeling Language*) anotados com restrições OCL (do inglês, *Object Constraint Language*). Essa abordagem aplica métodos de validação *lightweight* através da estratégia da hipótese do escopo reduzido (do inglês, S*mall Scope Hypothesis*) proposta por (JACKSON, 2006) aliada ao

método de validação de modelos conceituais utilizando ASSL (do inglês, *A Snapshot Sequence Language*), proposto por Gogolla e Richters (2005).

Diante desse contexto, o principal objetivo deste trabalho é responder a seguinte questão:

- De que maneira é possível aplicar as técnicas de validação de um modelo conceitual do domínio para geração de casos de teste independentes de plataforma para um Sistema Sob Teste (SST), afim de que seja atingida uma cobertura de testes satisfatória utilizando *software* livre?

A partir da questão principal também foram definidas algumas questões secundárias:

- Existem pesquisas que apresentam métodos para geração de casos de teste funcionais independentes de plataforma a partir da validação de modelos de classes UML enriquecidos com restrições OCL no contexto da TBM?
- De que maneira os métodos propostos são empregados?

A questão principal é respondida através da utilização da metodologia de pesquisa envolvendo um quase-experimento simples e um exemplo de aplicação. Já as questões secundárias são respondidas através do levantamento exploratório e mapeamento sistemático da literatura.

1.5 Metodologia

Para a proposta e desenvolvimento de uma abordagem para geração de casos de teste a partir da validação do modelo conceitual UML/OCL, algumas metas foram estabelecidas:

a) Apresentar um arcabouço conceitual referente ao tema proposto;
b) Identificar possíveis trabalhos correlatos através de um mapeamento sistemático da literatura;
c) Entender o funcionamento interno da ferramenta USE (do inglês, (UML-*based Specification Environment*);
d) Avaliar a aplicabilidade da hipótese do escopo reduzido em relação ao trabalho através de um quase-experimento simples;
e) Criar meios para aplicação do método de validação de modelos ASSL em conjunto com a hipótese do escopo reduzido para geração de casos de teste;

f) Criar um exemplo de utilização, verificar e discutir os resultados;

g) Identificar possíveis trabalhos futuros.

1.6 Organização do Trabalho

Esta dissertação está organizada em seis capítulos. O capítulo 1 mostrou a introdução do trabalho, ressaltando a sua contextualização, motivação, o problema tratado, e seus objetivos. O capítulo 2 discorre sobre o referencial teórico deste trabalho, através de um resumo da revisão bibliográfica e dos trabalhos relacionados. O capítulo 3 aborda a aplicabilidade da hipótese do escopo reduzido em relação ao trabalho. O capítulo 4 apresenta a abordagem proposta através de um exemplo de utilização. O capítulo 5 apresenta uma discussão sobre os resultados obtidos após a utilização da metodologia. O capítulo 6 conclui a dissertação, apresentando as considerações finais, as principais contribuições e sugestões de trabalhos futuros. Em seguida, é apresentada a lista de referências bibliográficas utilizadas. Finalmente, são apresentados na forma de apêndices os instrumentos utilizados no trabalho.

2 Referencial Teórico

Este capítulo trata da fundamentação teórica da dissertação. Ele está dividido em sete seções: a seção 2.1 introduz os conceitos fundamentais sobre Teste de *Software*; a seção 2.2 descreve os princípios relacionados ao TBM; a seção 2.3 faz referência ao uso das Regras de Negócio no contexto do desenvolvimento de *Software*; a seção 2.4 apresenta aspectos sobre a linguagem OCL, utilizada para descrever restrições sobre modelos UML; a seção 2.5 apresenta conceitos a respeito da ferramenta USE; a seção 2.6 enfatiza o uso da linguagem ASSL como forma de criação de cenários de validação de modelos UML e, por fim, a seção 2.7 apresenta os trabalhos relacionados a esta pesquisa.

2.1 Teste de *Software*

Teste de *software* é uma atividade do processo de desenvolvimento que consiste na execução controlada de programas cujo objetivo é verificar se o sistema atende ao que foi proposto, de modo a garantir que o produto final esteja dentro dos padrões da qualidade desejada (DIAS-NETO, 2008; SOMMERVILLE, 2011; PRESSMAN, 2011). A execução dos programas de teste visa expor os problemas pertencentes ao sistema para que se possa corrigi-los e aumentar o seu nível de confiança (DIAS-NETO, 2008). Dessa forma, a atividade de teste consiste na execução de um sistema a fim de que erros sejam detectados (UTTING e LEGEARD, 2007).

A busca por qualidade de um *software* se dá através de processos de V&V do *Software*. Enquanto a primeira tem por objetivo avaliar o *software* quanto a sua especificação documentada, a segunda objetiva avaliá-lo tendo como partida o ponto de vista e necessidades do usuário (PRESSMAN, 2011; SOMMERVILLE, 2011).

O processo de verificação é composto por dois tipos de classificações distintas, sendo uma estática e outra dinâmica (SILVA-DE-SOUZA, 2012). Enquanto que verificações estáticas avaliam aspectos documentais do *software*, verificações dinâmicas atuam sobre o *software* em tempo de execução. É fundamental que os testes sejam realizados durante

todo o ciclo de vida do *software*, pois é dessa maneira que falhas e imprevistos são encontrados, aumentando a confiabilidade e qualidade do produto, minimizando eventuais riscos a que todo projeto está inerente (SOUSA, 2009). Segundo (DIAS-NETO, 2008), as atividades de teste são divididas em seis elementos:

- Casos de Teste: representam um modelo de teste que contém valores de entrada, que após executados retornam uma saída, que será avaliada em razão de uma condição de guarda e do seu resultado esperado;
- Procedimentos de Teste: dizem respeito aos passos necessários para execução dos casos de Teste;
- Critérios de Teste: objetivam aumentar o nível de confiança do produto através da seleção sistematizada dos dados a serem utilizados no teste;
- Critérios de Cobertura de Teste: especificam um nível de confiança ao qual um critério de teste deve estar aderente;
- Critérios de Adequação de Casos de Teste: verificam se um caso de teste ao ser executado é suficiente para testar determinada funcionalidade de um produto;
- Critério de Geração de Casos de Teste: Define as regras para geração dos casos de teste com base no critério de adequação.

De acordo com (DIJKSTRA, 1979), não é possível dizer que um *software* está livre de erros, de modo que um teste nunca provará a ausência de defeitos em um sistema. Para (SOUSA, 2009), defeitos estão ligados ao universo físico e são provocados pela interpretação equivocada do desenvolvedor, enquanto que o erro é a manifestação do defeito desenvolvido, e falha é o momento no qual o *software* tem comportamento diferente do esperado por causa de um erro gerado. Entretanto, é importante mencionar que mesmo que um *software* esteja repleto de erros, este poderá nunca vir a apresentar uma falha (SILVA-DE-SOUZA, 2012). A Figura 1 mostra a relação entre defeito, erro e falha.

Figura 1: Defeito, Erro e Falha (DIAS-NETO, 2008)

O processo de teste pode ser dividido em três etapas: preparação do teste, execução do teste e registro do teste (IEEE, 1998). Durante a etapa de preparação do teste é observado a fase de planejamento onde são definidas três dimensões que objetivam saber o tipo de teste a ser realizado, a técnica utilizada para efetuá-lo e quando realizá-lo (SOUSA, 2009).

Os Tipos de Teste representam o que deve ser testado no sistema, isto é, quais as características do *software* que serão testadas. Podem ser tanto necessidades funcionais quanto não-funcionais. Já as técnicas de teste representam como o sistema deverá ser testado e subdivide-se em duas: caixa branca e caixa preta. A primeira se trata de uma técnica estrutural que leva em conta o código fonte do sistema, ou seja, o analista de teste tem acesso direto ao código, enquanto que a técnica de caixa preta se trata de um teste funcional, onde as verificações e validações são realizadas de acordo com os requisitos do sistema. Em níveis de teste é observado o momento no qual o sistema deverá ser testado. As atividades ocorrem em diferentes níveis do desenvolvimento e contém um teste especifico a ser executado (SOMMERVILLE, 2011). A Figura 2 apresenta as dimensões de teste encontradas.

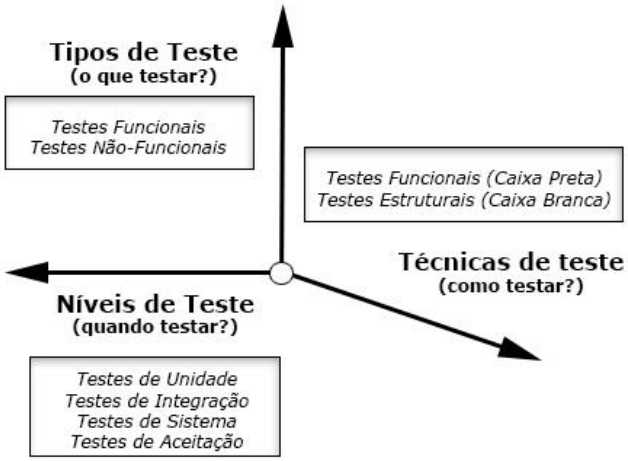


Figura 2: Dimensões do Teste de Software (UTTING e LEGEARD, 2007)

2.1.1 Tipos de Teste

Os tipos de teste indicam quais características do produto deverão ser testadas, seja do ponto de vista funcional ou não-funcional. Com relação aos tipos de teste funcionais destacam-se o teste de funcionalidade, que visa identificar possíveis defeitos relacionados às regras do negócio as quais o sistema deverá estar de acordo; o teste de regressão, que busca identificar possíveis inconsistências após o tratamento de defeitos anteriores; e o teste de usabilidade, que trata da operacionalidade do *software* por parte dos usuários. Já os testes não-funcionais estão relacionados a aspectos de desempenho, confiabilidade e segurança do produto em seu ambiente de execução, tais como velocidade de processamento adequada, carga de dados suportada e acesso restrito a determinadas funções do *software* (SOMMERVILLE, 2011).

2.1.2 Níveis de Teste

Os níveis de teste estão diretamente ligados ao momento no qual um teste deverá ser realizado, isto é, ocorrem de maneira concomitante ao desenvolvimento do *software* (SILVA-DE-SOUZE, 2012). A Figura 3 exemplifica a relação entre as fases do desenvolvimento do sistema e do teste a ser realizado.

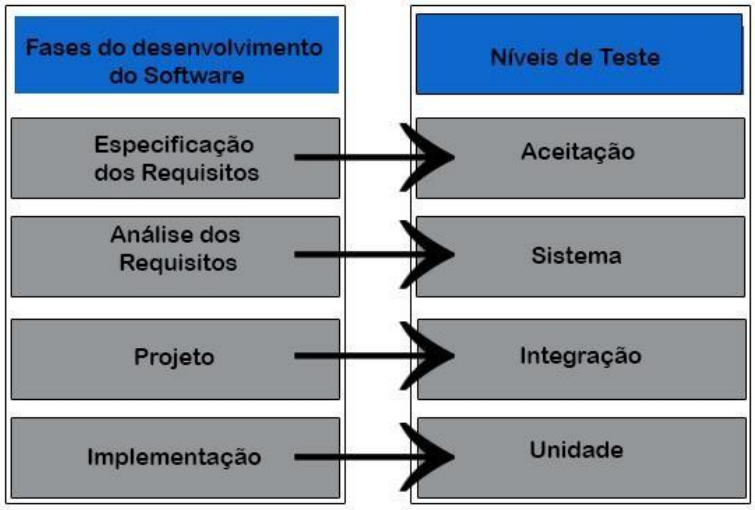


Figura 3: A fases do desenvolvimento do *Software* e os níveis de teste

Cada nível de teste abrange um conjunto de testes específicos a determinada fase do ciclo de desenvolvimento do *software.* Esses níveis são descritos como:

- Teste de Unidade: Nesta fase os testes são projetados de maneira a investigar possíveis presenças de defeitos em uma unidade ou modulo mínimo do *software.* Cada unidade é testada de maneira independente das demais partes do sistema.

- Teste de Integração: O objetivo do teste de integração é verificar o comportamento do sistema à medida que relacionamentos são criados entre as suas unidades.

- Teste de Sistema: Testa o sistema como um todo de modo a verificar se todas as partes estão se comportando da maneira esperada quando executadas em conjunto.

- Teste de Aceitação: Verifica se o sistema atende aos requisitos especificados pelo cliente. Nessa fase o objetivo é validar se o *software* atende as necessidades e expectativas dos usuários.

2.1.3 Técnicas de Teste

As técnicas de teste dizem respeito à maneira como as informações são obtidas para que um teste seja planejado e executado. Elas estão classificadas em Técnica Funcional e Técnica Estrutural, conforme especificado por (PRESSMAN, 2011):

- Técnica funcional ou caixa preta: nessa técnica o código fonte do sistema não é acessado pelo analista responsável pelos testes. Nesse caso os testes são inteiramente projetados com relação à especificação formal dos requisitos, ou seja, os detalhes da implementação do *software* são desconhecidos e não interferem no modo como o teste é projetado.

- Técnica estrutural ou caixa branca: Essa técnica visa compreender aspectos lógicos internos dos programas, ou seja, nessa fase objetiva-se aplicar testes, quando o código fonte é conhecido, com o intuito de avaliar o comportamento da implementação com relação aos dados de entrada e saída.

Em sistemas orientados a objetos, a técnica funcional pode ser aplicada aos requisitos especificados em formas de diagramas da *Unified Modeling Language* (UML). As informações relevantes aos testes são obtidas de modelos que representam a estrutura e o comportamento do sistema no ponto de vista funcional, seja de maneira estática ou dinâmica. Desse modo, um sistema bem modelado garante uma cobertura completa das funcionalidades do *software*, permitindo que todos os tipos de defeitos funcionais sejam encontrados. No entanto, essa cobertura completa pode gerar um crescimento exponencial de casos de teste, dificultando o processo de execução dos mesmos, tendo em vista que, segundo Sommerville (2011), testes exaustivos são impossíveis.

2.1.4 Critérios de Teste

Para contornar o problema da inviabilidade de geração de todas as combinações de entradas possíveis para um domínio, a técnica funcional de teste de *software* apresenta alguns critérios que visam diminuir o número de informações tratadas em uma busca por defeitos. Tais critérios de seleção são especificados com relação aos dados de entrada para o teste, onde o analista pode definir faixas de valores para testes positivos e negativos. Conforme Pressman (2011), os principais critérios de seleção são:

- Em particionamento em classes de equivalência os dados da especificação devem ser separados em classes de dados referentes aos valores válidos e classes referentes aos valores não válidos. Dessa forma, os testes são realizados com valores representativos, isto é, qualquer valor que esteja dentro da faixa de dados válidos deve obter um

resultado esperado válido, enquanto que qualquer valor dentro de uma faixa de valores inválidos deve apresentar uma falha como resultado da execução do procedimento de teste.

- Análise valor-limite trata dos valores de fronteira da aplicação, ou seja, esse critério se baseia nos valores mínimos de sucesso e falha. Quando utilizado juntamente ao particionamento de equivalência, este critério é seleciona os valores mínimos de cada classe de dados.

- Grafos de causa-efeito representam uma tabela de decisão onde um determinado objeto ou ação provoca um efeito a ser avaliado. É um grafo que estabelece a relação entre o efeito (ações esperadas) e suas causas (dados de entrada).

2.1.5 Casos de Teste

Um caso de teste descreve uma situação de teste contendo dados de entrada, condições de execução e resultado da saída esperada, que é provida por um oráculo (IEEE-829, 1998). Uma definição mais simples é apresentada por Naik e Tripathy (2008), que apresenta um caso de teste apenas como uma dupla contendo <*dados de entrada*> e <*saída esperada*>. Os casos de teste podem ser classificados como positivo ou negativo. Um caso positivo é aquele que é projetado de forma a verificar se a resposta do sistema está de acordo com aquela especificada nos requisitos. Um caso de teste negativo, por outro lado, é projetado para verificar a resposta do sistema a situações que estão fora da especificação e, portanto, devem retornar uma situação de erro (SILVA-DE-SOUZA, 2012; SPILLNER, LINZ e SCHAEFER, 2014). Casos de teste negativos também estão ligados ao teste de robustez do sistema, uma vez que é necessário que o sistema seja capaz de lidar com situações não previstas (dados de entrada não esperados) sem que haja interrupção completa de suas funcionalidades (SPILLNER, LINZ e SCHAEFER, 2014; NAYAK e TRIPATHY, 2008).

No contexto deste trabalho, um caso de teste foi definido como uma dupla (*Estado*, *Situação*), onde estado é um conjunto de objetos e associações (*snapshot*) do modelo conceitual e situação representa a avaliação desse estado. Um estado é avaliado como válido quando não viola nenhuma das restrições de integridade do modelo e inválido no caso contrário.

2.2 Teste Baseado em Modelo

A criação manual de testes é uma tarefa que consome tempo e recursos durante o desenvolvimento do *software*. Além disso, por se tratar de uma tarefa que demanda alto conhecimento do analista de testes a respeito do domínio, está sujeito a erros interpretativos, aumentando a probabilidade de que haja defeitos no sistema, ocasionando menor garantia de qualidade final ao produto (SILVA-DE-SOUZA, 2012; SOUSA, 2009). Segundo Utting e Legeard (2007), o processo de teste em *software* demanda de 30% a 60% do esforço gasto com recursos utilizado durante o ciclo de desenvolvimento do produto. Para contornar esses gastos, a indústria ao longo dos anos vem optando por soluções que automatizem o processo de teste de *software*, principalmente no que diz respeito a sua execução. É nesse contexto que surge o conceito denominado Teste Baseado em Modelos (TBM), segmento da engenharia de *software* que apresenta meios para automatização da geração de testes a partir de modelos de domínio (UTTING e LEGEARD, 2007).

As abordagens TBM são uma forma de reduzir os custos com o processo de teste durante o desenvolvimento de *software*. Ao optar por construir testes a partir de modelos que expressem o comportamento dinâmico do sistema, aumenta-se a garantia de cobertura dos testes, desde que esses modelos estejam bem construídos (UTTING e LEGEARD, 2007). De acordo com Utting e Legeard (2007), existem quatro tipos de abordagens relacionadas ao Teste Baseado em Modelos:

- Geração de dados de entrada a partir de um modelo de domínio: nessa abordagem são providos meios e técnicas para geração de dados de entrada a partir de um subconjunto de informações providas pelo modelo. Nesse caso, os valores são selecionados e combinados a fim de que valores de entrada sejam gerados com base em algum critério de teste, como particionamento em classes de equivalência, análise do valor limite ou teste em pares, por exemplo;

- Geração de casos de teste a partir de um modelo do ambiente: o objetivo desta abordagem é gerar os testes a partir de um modelo que expresse o ambiente do sistema sobre teste;

- Geração de casos de teste com oráculos a partir do comportamento do domínio: essa abordagem traz o conceito de resultado esperado. Nela, testes executáveis são gerados a partir de um modelo que expresse o comportamento do sistema de modo que seja possível obter não só os valores de entrada como também comparar o resultado da saída com o resultado esperado para o teste;

- Geração de *scripts* de teste executáveis a partir de testes abstratos: nessa abordagem, cenários abstratos provenientes de modelos de domínio são transformados em *scripts* de teste para uma determinada plataforma específica. É necessário que seja implementado uma transformação entre modelos independentes de plataforma e um modelo dependente de plataforma.

Apesar das quatro abordagens terem grande importância em TBM, é a terceira abordagem que traz maior valor para o teste de *software*, pois a partir de um modelo comportamental do sistema sobre teste é possível predizer o comportamento esperado para determinadas situações a partir dos relacionamentos entre os valores de entrada e saída. Dessa maneira é possível predizer o resultado de um teste após a sua execução (UTTING e LEGEARD, 2007). A figura 4 apresenta, de maneira simplificada, um exemplo do processo de geração de testes utilizando as abordagens TBM e Manual.

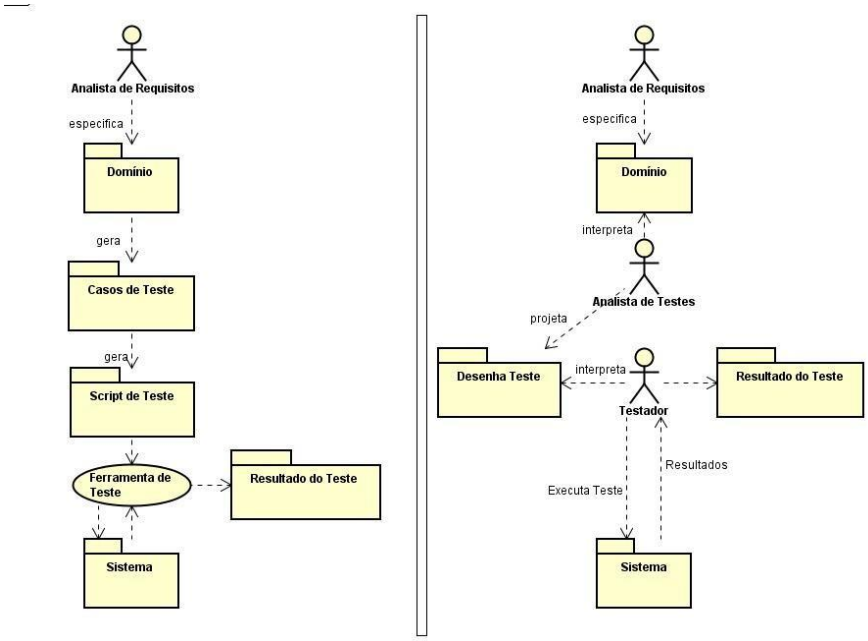


Figura 4: Abordagem baseada em modelos e Manual

O conceito de predição do resultado esperado é denominado de: problema do oráculo. Em abordagens manuais, oráculo é o analista que visualiza o resultado de um teste após sua execução e diz se um teste teve sucesso ou se falhou (PETERS e PARMAS, 1994). Algumas abordagens surgiram a fim de solucionar o problema do oráculo na automatização dos resultados de testes providos a partir de modelos. Segundo Earll *et al*. (2014), os oráculos estão divididos em quatro categorias:

- Oráculos de testes de especificações são definidos a partir de modelos matemáticos derivados a partir de um modelo comportamental do sistema. *Design by Contract* é uma maneira de solucionar o problema do oráculo a partir de pré e pós-condições associadas às operações do modelo. Outra forma de utilização de oráculos nesta abordagem é através da violação de invariantes presentes no modelo;

- Oráculos de testes derivados são extraídos a partir de diversos artefatos do sistema, bem como documentação textual e execução, seja do sistema atual como de versões anteriores, podendo ser utilizado em testes de regressão por exemplo. Da mesma maneira que oráculos especificados, também podem ser utilizados a partir de invariantes;

- Oráculos de teste implícitos se baseiam em conhecimentos gerais, isto é, não é necessário entender o modelo e nem a sua especificação. São testes que têm como base algo que é de entendimento geral, como por exemplo falhas ocorridas devido a *stack overflow* ou qualquer processo que tenha sua execução terminada repentinamente. Um exemplo de técnica para esse tipo de teste é a de *Fuzzing,* onde valores de entrada são gerados aleatoriamente a fim de que o sistema falhe;

- Oráculos não-automatizados representam os testadores que após executarem um teste avaliam e catalogam o seu resultado com base em uma especificação formal de um caso de teste projetada por um analista de teste.

Um modelo conciso que represente todo o comportamento do sistema garante uma cobertura completa do sistema, no entanto, isso pode causar um número

exponencial de valores de entrada, inviabilizando a geração desse tipo de abordagem. Devido a esse problema, a TBM define alguns critérios de cobertura para seleção dos testes como: fluxo de dados, controle de fluxo e fluxo de transição. Esses critérios podem ser usados em diversos diagramas da UML. Em geral, os algoritmos para seleção dos testes com base nesses critérios são realizados com relação a uma busca por um caminho, seja parcial, total ou em *loop,* e a uma ou mais condições. Além desses critérios de seleção citados anteriormente, existem outros critérios específicos baseado em UML para três tipos de diagrama: classes, objetos e sequência (UTTING e LEGEARD, 2007).

Os critérios de seleção que se baseiam em diagramas de classes foram inicialmente pensados para serem utilizados em testes para validação dos modelos, porém podem ser utilizados para criação de casos de teste para um sistema em funcionamento (UTTING e LEGEARD, 2007). Este critério divide-se em três tipos:

- Baseado em multiplicidade (ou relacionamentos): nesse critério são definidos quantos objetos de um tipo de classe podem estar ligados a outro objeto. Torna-se então necessário que cada par de um relacionamento seja criado para um teste. Neste caso, todas as instancias referentes ao par de relacionamentos são criadas seguindo alguma estratégia de seleção, como por exemplo particionamento em classes de equivalência;

- Generalização: define que para cada superclasse do modelo, pelo menos um objeto de sua subclasse deve ser instanciado;

- Atributo: para todo atributo de uma classe do modelo, gerar um conjunto de valores que permitam gerar todas as combinações possíveis seguindo alguma determinada estratégia de seleção, como análise valor limite por exemplo.

2.3 Regras de Negócio

As regras de negócio visam controlar os aspectos internos e externos à organização, no que diz respeito à maneira como os negócios são estruturados e operacionalizados (CUNHA, 2009). As regras de negócio definem pré e pós-condições aplicáveis às operações que devem ser verdadeiras em determinadas situações (ARAUJO, 2010). Desta maneira, elas são parte fundamental do teste em *software*, tendo em vista que definem ou restringem algum ponto de vista do negócio, seja no âmbito organizacional ou no âmbito dos sistemas da informação (CUNHA, 2009). No lado dos sistemas de informação, as regras de negócio são vistas como parte dos requisitos do sistema, de modo que seja possível projetá-las, implementá-las, testá-las e analisá-las; enquanto que no lado do negócio, as regras são vistas como diretivas responsáveis por conduzir ou influenciar o comportamento do próprio negócio a qual a organização está investindo (CUNHA, 2009).

Existe uma série de características inerentes às regras de negócio que dizem respeito à atomicidade, objetividade, relacionamentos e independência com relação à tecnologia. Além disso, as regras de negócio, no âmbito dos sistemas da informação, apenas dizem como é o comportamento esperado para o produto, mas não como implementá-lo (CUNHA, 2009).

A execução de um sistema da informação está diretamente ligada às operações de criação, atualização, recuperação e exclusão dos objetos de negócio. Tais objetos podem ser considerados como instâncias de classes, sejam elas abstratas ou concretas. Dessa maneira, um diagrama de classes da UML pode ser entendido como uma representação dos objetos de negócio, das suas características, bem como suas possíveis restrições, e de seus relacionamentos em um modelo independente de plataforma (ARAUJO, 2010). A Figura 6 exemplifica um fragmento metamodelo dos diagramas de classes da UML definido pelo OMG (do inglês, *Object Management Group*) (OMG, 2014):

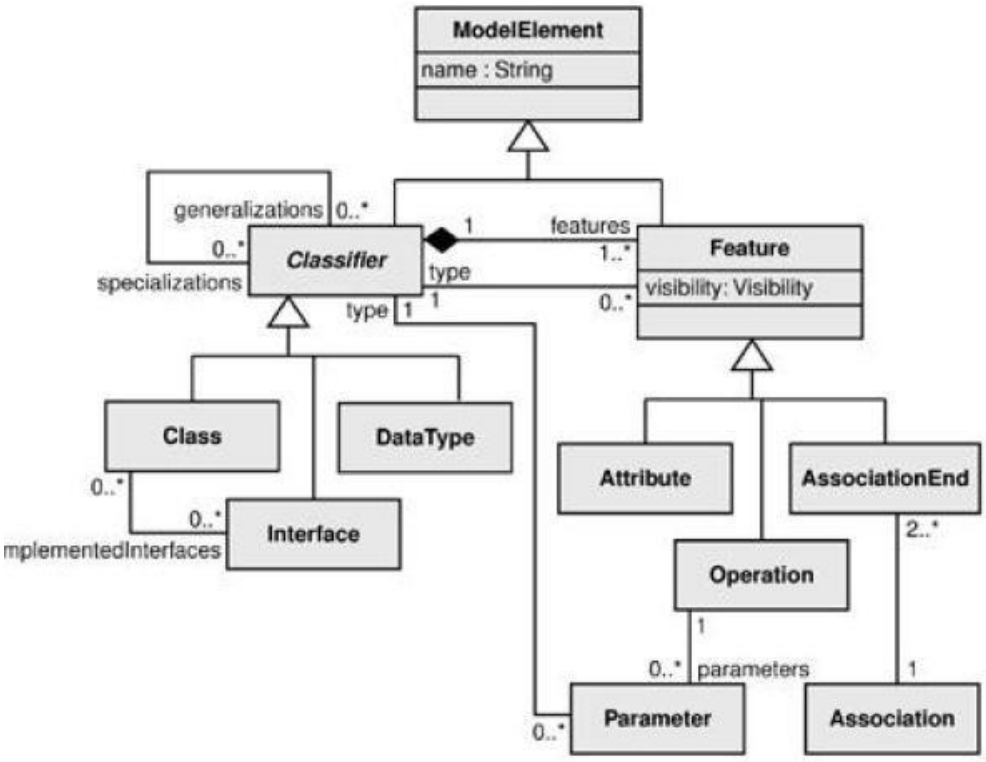


Figura 5: Metamodelo Simplificado do Diagrama de Classes da UML (ARAÚJO, 2010)

Em um modelo de classes UML, as regras de negócio podem estar representadas de três maneiras: através dos atributos das classes, relacionamentos entre as classes e através de especificação OCL. Os relacionamentos entre as classes definem as regras de multiplicidade (ou cardinalidade) entre os objetos do domínio, enquanto que as regras especificadas em OCL definem restrições em formas de invariantes, geralmente sobre atributos do modelo, e pré e pós-condições sobre as operações de uma classe.

2.3.1 Restrições de Multiplicidade

As regras de negócio que definem os relacionamentos entre os objetos de um domínio dizem respeito ao conceito de restrições de integridade. É através da multiplicidade expressa nas associações que é possível gerar testes que busquem por situações que violem a integridade referencial dos objetos do sistema (SILVA-DE-SOUZA, 2012; OLIVÉ, 2007). A Figura 7 apresenta um exemplo de restrição de multiplicidade expressa em um relacionamento.

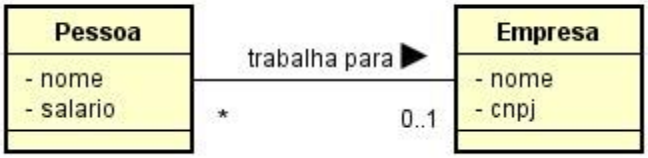


Figura 6: Restrição de integridade entre duas classes do sistema

No contexto apresentado na figura existe uma restrição de integridade de um {1} para zero ou um {0..1}, onde está subentendido que um objeto da classe *Pessoa* só pode se relacionar com no máximo um objeto da classe *Empresa*, isto é, existe uma regra de negócio especificada que diz que uma pessoa só pode trabalhar para uma única empresa em um dado instante do tempo. Em geral, os valores das multiplicidades indicam um limite mínimo e um limite máximo para os objetos de um relacionamento. A tabela 1 apresenta as seguintes restrições de multiplicidade impostas pela UML 2.0.

Tabela 1: Tipos de Multiplicidade

Multiplicidade	**Descrição**
{0..1}	No máximo um. Indicam que um objeto não precisa estar necessariamente relacionado a outro objeto, porém ele não pode ficar ligado a mais do que um objeto.
{1..1}	No mínimo um e no máximo 1. Neste caso um objeto só existe se houver um único objeto relacionado a ele.
{0..N}	Muitos. Não há restrição quanto ao número de objetos em um relacionamento.
{1..N}	No mínimo um. Indica que há pelo menos um objeto obrigatório na associação.
{2..4}	Valores específicos. Indicam o valor mínimo e máximo de objetos possíveis em um relacionamento.

Deste modo, um sistema deve prover meios para que esses tipos de regras não sejam violadas. A figura 7 apresenta um diagrama de objetos inconsistente, que deverá apresentar uma falha no sistema caso o relacionamento seja de {0..1}.

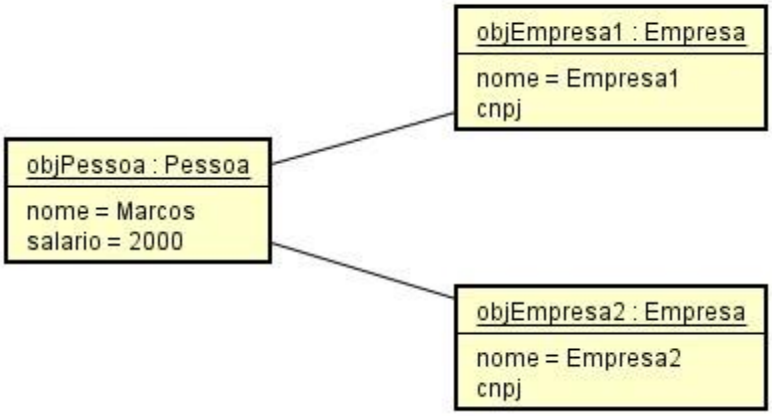


Figura 7: Exemplo de restrição de integridade violada

Como a multiplicidade especificada no modelo de exemplo é de {0..1}, então um objeto da classe *Pessoa* só poderia estar associado a no máximo um objeto da classe *Empresa*. Essa situação denota um exemplo de violação da integridade referencial do modelo (OLIVÉ, 2007). Outros exemplos de violação dessa restrição estão ligados às operações CRUD (acrônimo termos em inglês: *Create*, *Read*, *Update* e *Delete*), onde é necessário verificar se tais regras não foram quebradas após as operações de criação, atualização e remoção de objetos do sistema (SILVA-DE-SOUZA, 2012).

Além das regras expressas através da multiplicidade nas associações, existem também outros tipos de relacionamentos que indicam o mesmo tipo de regras de negócio, como por exemplo: agregações e composições. Esses tipos de associações indicam o conceito de *parte-todo*, no qual objetos de um tipo fazem parte de um objeto considerado o todo da relação. Agregações não são mutuamente exclusivas, pois objetos pertencentes à parte de um todo podem estar relacionados a outros objetos a qualquer instante do tempo, diferentemente do tipo Composição, onde o objeto da *parte* é exclusivamente relacionado ao *todo* (OLIVÉ, 2007).

2.4 *Object Constraint Language* (OCL)

A OCL (OMG, 2014) é uma linguagem textual de restrição de objetos definida pelo OMG, cujo objetivo é especificar regras que não podem ser definidas graficamente em um modelo aderente ao padrão MOF (do inglês, *Meta-Object Facility*[1]). Desse modo, a OCL permite que as regras de negócio sejam expressas em uma linguagem computacional como sendo um complemento sobre os aspectos de diagramação da UML, onde os principais propósitos da linguagem são: especificar restrições através de invariantes, especificar pré e pós- condições sobre as operações, condições de guarda e regras de derivação. Além disso, a OCL permite tornar modelos mais condizentes com a realidade e servir como linguagem de consulta para modelos (OMG,2014).

A OCL pode ser definida como uma linguagem de especificação de restrições e consultas (WARMER e KLEPPE, 2003). Tais restrições são especificadas através de expressões associadas a elementos do modelo, onde cada uma representa valores ou objetos. Cada restrição está associada a um termo do negócio, de modo que suas sentenças estejam divididas em quatro partes: contexto, propriedade, operação e palavras reservadas (ARAUJO, 2010). Essas partes estão definidas no metamodelo OCL através de dois pacotes: pacote de tipos e pacote de expressões. A Figura 8 mostra um fragmento do metamodelo OCL, referente ao pacote de tipos.

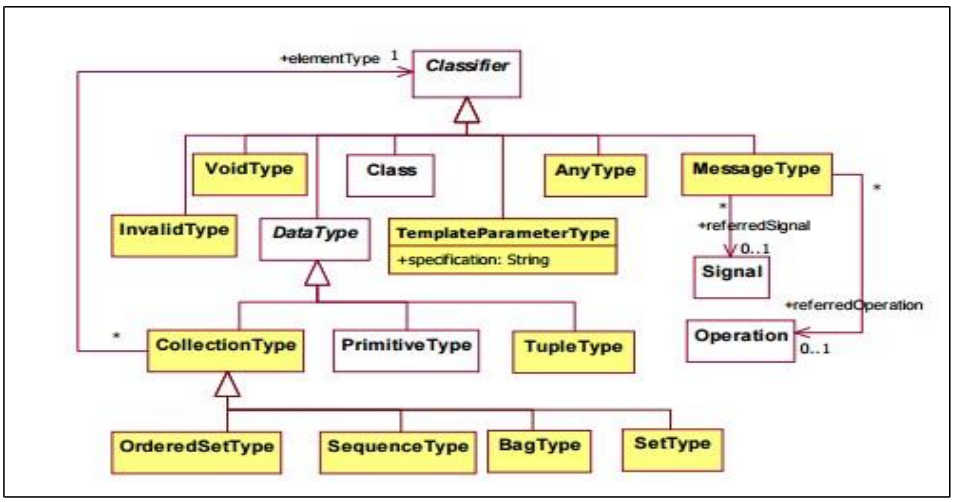


Figura 8: Modelo abstrato do pacote de tipos da OCL (OMG, 2014)

[1] é um padrão, também adotado pelo grupo OMG, que define todos os conceitos que podem ser utilizados em uma linguagem, de forma independente da tecnologia.

O pacote de tipos da OCL define aspectos relacionados às regras de tipos dos objetos da linguagem, isto é, cada expressão OCL é definida por um tipo especifico que pode ser declarado de maneira direta ou através de derivação. O tipo básico representado pelo metamodelo é denominado *Classifier*, que representa todos os demais tipos subtipos de classificadores UML. Essa parte da sintaxe abstrata da OCL também herda do metamodelo UML as estruturas de *Classes*, *Tipo de Dados*, *Operações* e *Signals* (OMG, 2014). Dentre os tipos específicos apresentados no metamodelo, destacam-se:

- *VoidType*: é a metaclasse do tipo *OclVoid*, que está em conformidade com todos os tipos não inválidos da OCL. Tipos *OclVoid* apresentam uma única possibilidade de instância, denominada de *null,* representando a não presença de valor.

- *AnyType*: representa a metaclasse do tipo *OclAny.* É um tipo especial que está em conformidade com todos os demais tipos OCL e *Classificadores* UML, como as classes e os tipos de dados primitivos.

- *MessageType*: descrevem as mensagens OCL. Representam uma referência às operações ou sinais, sua instância é denominada *OclMessages*.

- *CollectionType*: descrevem uma lista de elementos de um determinado tipo. Está representado por quatro tipos: *OrderedSetType*, *SequenceType*, *BagType* e *SetType*. O tipo *OrderedSet* permite um conjunto de valores ordenados não repetidos, *Sequence* representa O tipo *Sequence* é uma coleção que permite múltiplos valores iguais e ordenados, diferentemente do tipo *Bag* que também permite a alocação de múltiplos valores, porém sem ordenação. Já o tipo *Set* permite um conjunto de valores não repetidos e não ordenados.

- *TupleType*: é uma estrutura que permite a definição de diferentes tipos em um único tipo de agregação. Não há nenhum tipo de restrição com relação aos valores que podem ser alocados na tupla.

O pacote de expressões do metamodelo UML define as regras referentes às estruturas que uma expressão deve satisfazer. A Figura 9 apresenta o metamodelo do pacote expressões OCL.

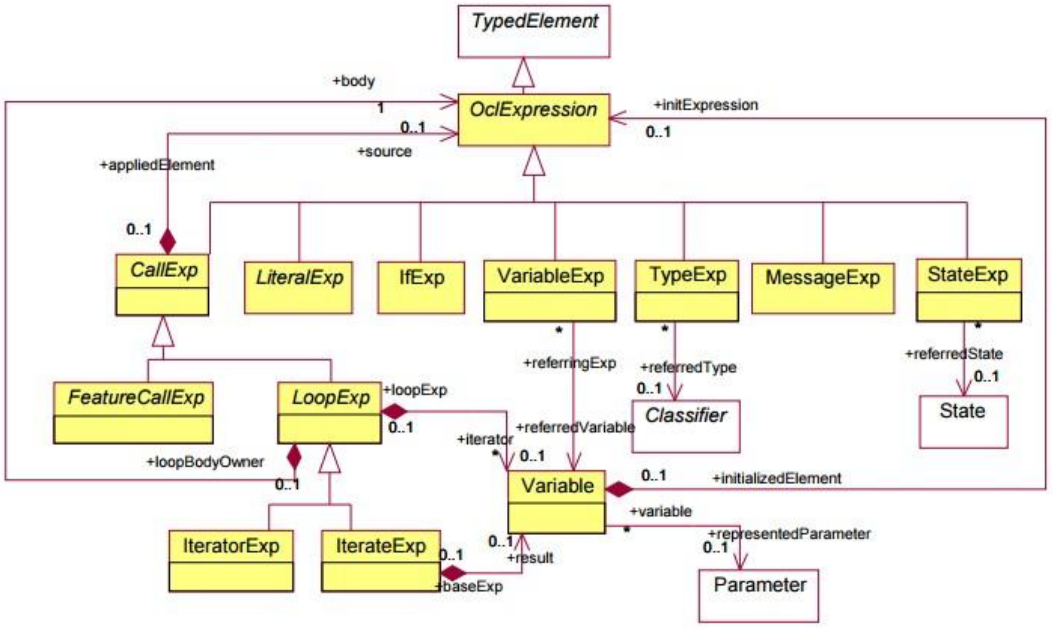


Figura 9: Sintaxe abstrata do metamodelo de expressões OCL (OGM, 2014)

Expressões OCL definem o comportamento esperado do modelo, isto é, definem as regras impostas pelo analista ao domínio modelado (CUNHA, 2009). Essas expressões seguem a estrutura definida pela sintaxe abstrata do metamodelo OCL, no qual sua estrutura principal se baseia nas seguintes metaclasses:

- *OclExpression*: é uma expressão que pode ser avaliada em um determinado contexto do tempo, onde cada expressão é caracterizada por apresentar um tipo específico, que pode ser determinado pelo seu contexto ou pela sua análise. A avaliação de uma *OclExpression* retorna um valor que pode ser usado como regras para invariantes, especificar consultas e iniciar atributos por exemplo;
- *CallExp*: metaclasse abstrata que se refere a uma característica ou propriedade de um *iterador* de coleções;
- *LiteralExp*: representa uma expressão que não possui argumentos, mas que retorna um valor. Representa através de suas especializações as classes *Boolean*, *String* e *Integer*;

- *VariableExp*: consiste de uma referência a uma variável, como *self*, *result* ou variáveis locais definidas pela palavra *"let";*
- *TypeExp*: Utilizada com as expressões *oclIsTypeOf*, *oclIsKindOf* e *oclAsType. É* responsável por um metatipo em uma determinada expressão;
- *FeatureCallExp*: representa a metaclasse de interligação com qualquer subtipo de *feature* do metamodelo UML relacionado aos atributos e operações.

Através de um conjunto de instruções OCL é possível especificar restrições a respeito da estrutura e do comportamento de um sistema (SILVA-DE-SOUZA, 2012). Tais instruções baseiam primordialmente em um contexto que contem invariantes e pré e pós- condições. A Figura 10 apresenta um exemplo simplificado de uma restrição OCL sobre a classe *Empregado*.

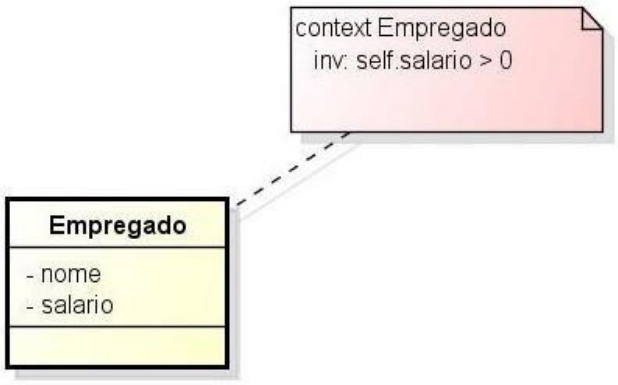


Figura 10: Exemplo simples de restrição OCL

No exemplo da figura 10 é apresentado uma classe empregado que possui nome e salário como atributos. A restrição de regra de negócio segundo o OCL expressado como comentário revela que uma instância da classe Empregado não pode ter salário menor do que zero. Essa restrição é especificada através de um invariante, que deverá sempre retornar um resultado verdadeiro durante o tempo de vida do objeto referenciado. Quando um invariante tem resultado falso, é dito que uma regra está sendo quebrada ou violada (WARMER e KLEPPE, 2003).

Além das restrições utilizando invariantes, a OCL permite o uso de pré epós-condições sobre determinada operação. A Figura 11 apresenta um exemplo simplificado de pré e pós-condições sobre uma operação.

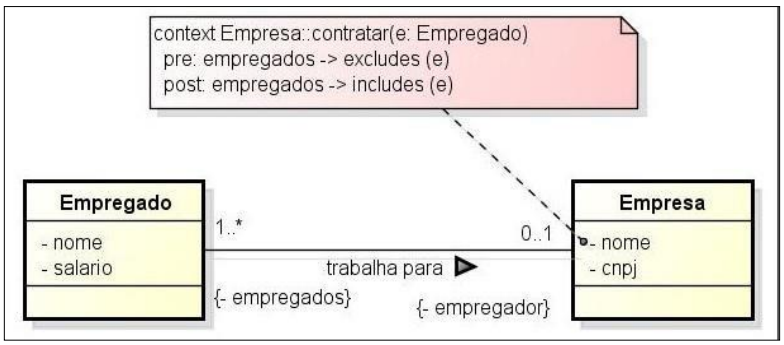


Figura 11: Pré e Pós condições sendo aplicadas a um modelo

O uso da OCL no exemplo na figura 11 demonstra que para uma Empresa contratar um Empregado é necessário que antes se satisfaça uma pré-condição, que é o fato de o empregado não estar empregado pela empresa. Em seguida o uso da palavra reservada *post* indica uma pós-condição, que garante que após a execução da operação contratar, a instância do empregado foi adicionada ao conjunto de empregados existentes.

As assertivas de pré e pós-condições definem a aplicabilidade e o efeito de uma operação sem especificar a sua implementação, isto é, não é preciso especificar como a operação deve ser feita, mas sim o que ela faz (WARMER e KLEPPE, 2003). Ao permitir que as regras de negócio sejam expressas computacionalmente em um modelo independente de plataforma, a OCL, juntamente com a UML, possibilita que casos de teste sejam gerados automaticamente (SILVA-DE-SOUZA, 2012).

2.5 UML-*based Specification Environment* (USE)

O USE é uma ferramenta utilizada para especificação e validação de sistemas da informação orientados a objeto utilizando UML e OCL. Ela tem como base a descrição textual do modelo de classes contendo classes, atributos, operações, relacionamentos e restrições OCL. O USE trata o modelo UML de maneira executável, fornecendo animações através de *snapshots,* que são cenários que representam um determinado estado do sistema em um instante do tempo. As animações geradas pela ferramenta permitem que o modelo seja validado pelo desenvolvedor (RICHTERS e GOGOLLA, 2005). A figura 12 demonstra a maneira de trabalho com a ferramenta.

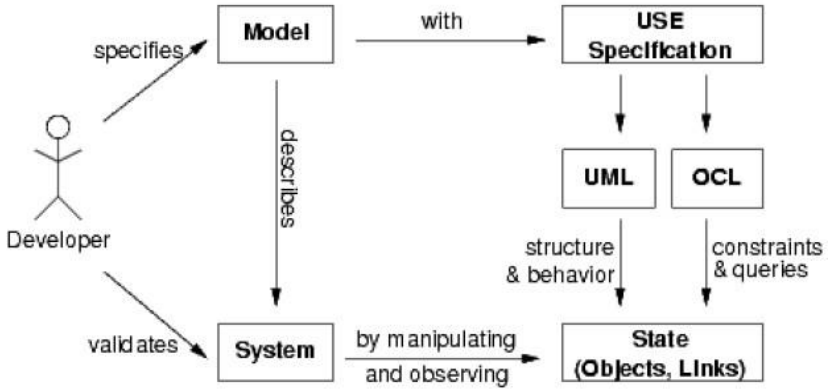


Figura 12: Visão geral da especificação através do USE (USE, 2007)

A arquitetura do USE é composta por duas camadas principais: descrição e interação. A camada de descrição diz respeito ao processamento e transformação da especificação textual em uma sintaxe abstrata que represente o modelo UML e OCL. Essa camada é responsável por gerar insumos para a camada de interação, que será responsável pela animação e interpretação das expressões OCL. Existem dois módulos principais na camada de interação, uma responsável pela geração de conteúdo dinâmico, como por exemplo operações CRUD sobre os objetos do estado, e outro módulo estático responsável pela validação do modelo (RICHTERS, 2002).

A validação dos modelos pode ser realizada de maneira manual ou semiautomática através de uma abordagem *lightweight*. Na maneira manual, o desenvolvedor cria o cenário desejado através da linguagem denominada de *Simple* OCL-*based Imperative Programming Language* (SOIL) ou pela própria interface do USE, em seguida executa o modelo e analisa a resposta gerada em busca de possíveis problemas. Já da maneira semiautomática é possível manipular os objetos de duas maneiras, por intermédio da linguagem ASSL ou do *plugin Model Validator*. O objetivo da validação de modelos é alcançar um bom projeto de especificação, que realmente satisfaça as necessidades dos investidores, antes que a fase de implementação comece (RICHTERS e GOGOLLA, 2005; GOGOLLA, BÜTTNER e RICHTERS, 2007).

2.6 *A Snapshot Sequence Language* (ASSL)

A OCL é uma linguagem funcional que não contém nenhum comando que altere o estado do sistema, na forma de criação, alteração ou remoção de objetos e associações. A ASSL surgiu com o intuito de permitir essa manipulação. Com a linguagem ASSL é possível criar e remover objetos, suas associações e seus atributos através de uma sequência de instruções imperativas. Cada conjunto de instruções é responsável por criar uma instância completa do modelo (ou parte dela) em forma de diagrama de objetos. A instância gerada pela ferramenta USE a partir de um *script* ASSL é denominada de *snapshot* e representa o estado do sistema em um determinado instante do tempo (GOGOLLA, BOHLING e RICHTERS, 2005). A figura 13 representa um exemplo de instrução ASSL.

```
procedure generatePersons(count:Integer)
var thePersons:Sequence(Person);
begin
thePersons:=CreateN(Person,[count]);
for p:Person in [thePersons]
  begin
  [p].name:=
    Any([Sequence{'Ada', 'Bob', 'Cher', 'Dan', 'Eva', 'Fred'}
    ->reject(n1|Person.allInstances.name->exists(n2|n1=n2))]);
  end;
end;
```

Figura 13: Exemplo de *script* ASSL

Uma *procedure* é o nome reservado da linguagem responsável por iniciar uma sequência de instruções. Ela contém uma assinatura e pode ter a presença de parâmetros. Dentro de uma *procedure* ASSL é possível especificar estruturas de condição (*if*) e repetição (*for*). As instruções da linguagem são classificadas em:

(1) Instruções que alteram o estado: são instruções relacionadas às operações de criação e remoção de objetos, relacionamentos e atribuição de valores sobre atributos;

(2) Instruções de valores aleatórios: permite especificar uma faixa de valores que serão associados a atributos ou relacionamentos de maneira aleatória;

(3) Instruções de transição múltipla: não alteram o estado final do sistema. Apesar de realizar manipulações alterando os cenários, as instruções de transição múltipla (*Try*) não definem um estado após o termino de sua execução;

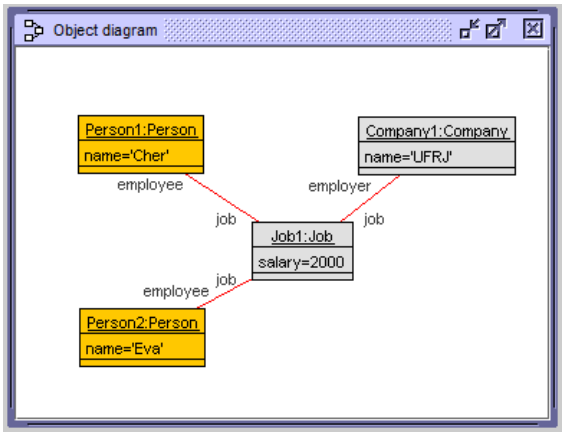


Figura 14: Exemplo de Snaptshot gerado

Além da manipulação dos cenários, o ASSL também permite a validação da especificação do modelo conceitual (Ver figura 14). Essa validação é feita por intermédio dos comandos *"gen start", "gen result" e "gen result accept"*. Esses comandos em conjunto permitem que o analista verifique se tais configurações de cenários especificados em ASSL satisfazem ou não as restrições do modelo. Quando um cenário com restrições violadas é encontrado (um contraexemplo), então o procedimento de validação é automaticamente interrompido pela ferramenta USE (KUHLMANN, HAMANN e GOGOLLA, 2011).

No contexto deste trabalho, foi necessário alterar o código fonte do USE para que essa interrupção não ocorra. Esta alteração permitiu a extração tanto dos casos de teste negativos quanto dos positivos a partir dos *snapshots* gerados.

2.6.1 Os Métodos de Validação *Lightweight*

Os métodos formais tradicionais (*heavyweight*) para validação de modelos conceituais envolvem a produção de uma prova matemática para verificar a corretude da especificação, tarefa que nem sempre é viável já que esse custo supera em muitas vezes o valor gasto com a especificação. Nesse contexto, observou-se a necessidade de um método de validação que obtenha os benefícios dos métodos tradicionais, mas com menor custo de execução (JACKSON e WING, 1996).

Os métodos *lightweight* utilizam características que objetivam aplicar técnicas de validação e análise não só para o sistema completo, como para partes dele. Dessa forma, não é necessário que a especificação esteja completa para que a validação ocorra, o que é ideal no ciclo de desenvolvimento do *software* (JACKSON E WING, 1996). Jackson (2006)

propôs a ferramenta Alloy, que aplica um método de validação *lightweight* através da hipótese do escopo reduzido. Essa hipótese sugere que sistemas que falham com grande número de instâncias quase sempre falhariam com um número pequeno de instâncias com propriedades similares. Assim, validando todos os casos com pequenos números de instâncias, nos permite ter uma certa confiança que, os casos com grande número de instancias também serão validados (JACKSON, 2006; GUIMARÃES *et al*., 2014).

2.7 Trabalhos Relacionados

Diversas abordagens relacionadas à geração de casos de teste já foram descritas na literatura; no entanto, não são muitas as propostas que fazem uso das funcionalidades da ferramenta USE como meio principal para geração dos testes. Além disso, pesquisas recentes indicam que boa parte dessas abordagens trata da geração de testes a partir de dois ou mais diagramas da UML em conjunto, principalmente os diagramas de sequência, atividades e estados.

Desta maneira, foi realizado um mapeamento sistemático com o intuito de reunir informações sobre a pesquisa em TBM a partir de modelos UML enriquecidos com restrições OCL. Esse mapeamento sistemático foi baseado no protocolo definido por Biolchini *et al*. (2005), buscando responder as seguintes questões:

- Existem pesquisas que apresentem métodos para geração de casos de teste funcionais independentes de plataforma a partir da validação de modelos de domínio de classes UML enriquecidos com restrições OCL dentro do contexto da TBM?
- Se existe, de que maneira os métodos propostos são empregados?”

O mapeamento sistemático foi realizado durante o ano de 2016 entre os meses de agosto e outubro, e teve como objetivo identificar os principais meios de geração de testes a partir de diagramas da UML enriquecidos com restrições OCL correlatos a este trabalho. Para tal, as seguintes *strings* de busca foram empregadas:

- *("model based test" or "model based testing" or "test model" or "model testing" or "model driven test" or "model driven testing" or "specification testing" or "requirements testing" or "test specification" or "requirements test" or "black box testing" or "black box test" or "test case generation" or "test case automation" or "test generation" or "test automation" or "test*

scenarios") and ("UML" or "unified modeling language" or "class diagram") and ("OCL" or "object constraint language").

As *strings* de busca foram empregadas em três bases de dados digitais: *Scopus Elsevier*, ACM *Digital Libray* e IEEE *Xplorer*. O *Google* Acadêmico (do inglês, *Google Scholar*) não foi utilizado neste mapeamento em razão da limitação de caracteres para a *string* de busca. Para a seleção dos artigos de interesse desta revisão foram aplicados os seguintes critérios de inclusão:

- Conteúdo principal sobre a utilização de UML e OCL para geração de testes de *Software* (correlatos a este trabalho).
- Os estudos devem estar disponíveis para *download* na Web.
- Disponível em inglês ou português.

O processo de seleção dos artigos foi realizado em três iterações: na primeira iteração foram avaliados os títulos, resumos e as palavras-chave de todos os trabalhos retornados pela ferramenta de busca. Em seguida, os artigos selecionados na primeira fase foram submetidos à segunda iteração, onde a introdução, metodologia e conclusão foram analisadas. E, por fim, a última iteração diz respeito à leitura completa dos artigos selecionados. A tabela 2 apresenta as etapas de seleção dos artigos encontrados e avaliados.

Tabela 2: Fases do processo de seleção dos artigos

Base de Dados	1ª Iteração	2ª Iteração	3ª Iteração
Scopus	71 artigos	18 artigos	17 artigos
IEEE	34 artigos	17 artigos	10 artigos
ACM	34 artigos	08 artigos	06 artigos

Após a leitura dos 33 artigos selecionados na terceira fase do processo de seleção, não foi encontrado artigos que tratem da geração de casos de teste independente de plataforma a partir da validação de seu modelo conceitual. No entanto alguns foram selecionados por apresentar algumas características correlatas a este trabalho.

O trabalho de Araújo (2010) apresenta um método *lightweight* para verificação de inconsistências entre processos e regras de negócio através da utilização de modelos UML anotados com restrições OCL. Os modelos de processos de negócio são executados como

cenários de teste através da utilização da ferramenta USE, de modo que é possível aos observar e identificar, de forma interativa, os efeitos produzidos pelas atividades do processo.

Já o trabalho de Francisco e Castro (2012) apresenta uma abordagem baseada na geração de valores aleatórios para as operações do diagrama de classes. O modelo UML/OCL é transformado em propriedades que podem ser utilizadas para geração de casos de teste independentes de plataforma através da ferramenta *QuickCheck*. A abordagem proposta pelos autores é dividida em dois métodos. Um método é referente às operações que não possuem *side-effect* e outro às operações que possuem *side-effect.* O diagrama UML é utilizado para conhecer as operações a serem testadas e os detalhes dos tipos de dados, valores e retorno da função a ser testada. Enquanto que as expressões OCL são usadas para geração do oraculo de teste através das pré e pós-condições sobre as operações.

O trabalho de Nayak e Samanta (2010) define uma abordagem para sintetizar dados de teste a partir de cenários gerados através dos diagramas de sequencia e diagramas classes com OCL. Neste caso, o diagrama de sequencia é enriquecido com atributos derivados do diagrama de classes, enquanto as restrições são derivadas do OCL para então ser transformado em uma estrutura de grafo direcionado composto. Dessa maneira, o algoritmo proposto, pelos autores, recebe como entrada um diagrama de sequencia e um diagrama de classe, e tem como saída um grafo direcionado. Esse grafo direcionado será a entrada para a geração dos cenários abstratos de teste, que são sequencias de mensagem.

Chang e Lin (2016) abordam, em seu atrabalho, a criação de um *Framework* que segue os principios da MDA (do inglês, *Model Driven Architecture*) para geração de casos de teste unitários caixa preta através de uma estrutura baseada em restrições. Essas restrições são definidas através da linguagem OCL. Os testes gerados pelo *framework* são voltados a execução de operações das classes do domínio. Esse trabalho basea-se no conceito de *Constraint Logic Graph* para gerar dados com base nos critérios de seleção de particionamento em classes de equivalencia e no gerenciamento do critério de cobertura do teste a partir das expressões OCL. E no conceito de programação em lógica com restrições para gerar os dados de entrada e avaliar o resultado de saída esperado através da utilização do problema da satisfação de restrições.

Em Bouquet *et al*. (2008) é apresentado uma abordagem para geração de testes funcionais através da ferramenta *Leirios Smart Testing*. A ferramenta faz uso dos diagramas de classes, objetos e estados. Segundo os autores, o objetivo do trabalho é demonstrar como a ferramenta pode ser aplicada no contexto da TBM, de modo que os diagramas de classes são responsáveis por definir a visão estática do modelo, os diagramas de objetos definem o estado inicial do sistema sob teste, enquanto que a maquina de estado é usada para definir o comportamento dinâmico do modelo através de condições de guarda, pré e pós-condições OCL.

O trabalho proposto por Gogolla *et al*. (2015) apresenta um método para geração de cenários e de casos de teste através do uso de diagrama de classes enriquecidos com restrições OCL. Gogolla e companheiros utilizam da criação de um termo denominado de classificador (ou classificação), onde os objetos do modelo são classificados conforme o critério de seleção de partição de equivalencia, isto é, conjunto de objetos ou estados semelhantes representam uma mesma partição de classe, criando cenários particionados equivalentes. Esse classificador é implementado juntamente ao *plugin Model Validator* da ferramenta USE. Essa extesão do USE, assim como o ASSL, permite verificar se um modelo está correto ou não, porém ao contrário do ASSL, ele faz uso da transformação do modelo UML/OCL em modelo relacional, aumentando a velocidade com que o modelo é validado. Nesse trabalho, os autores focam na geração de testes para transformações entre os modelos.

Em Bao-Lin *et al*. (2007) é apresentada uma abordagem para geração de testes a partir dos diagramas de sequência e classes anotados com OCL. Os autores denominam esse processo de *SOTest*, cujo algoritmo é composto por quatro partes:

(1) Conversão do diagrama de sequência em uma árvore de cenários, onde cada cenário é uma sequência de mensagens;
(2) Identificar todas as classes, operações e atributos relevantes de acordo com cada caminho da mensagem;
(3) Utilizar o diagrama de classe e restrições OCL para determinar os dados de entrada e gerar os casos de teste;
(4) Executar e validar os testes.

A proposta de Bizerra *et al*. (2012) trabalha com a geração de instâncias de teste funcionais a partir da utilização de gabaritos sobre as pré e pós-condições escritas com operações OCL em um modelo de classes UML. O método proposto pelos autores é divido em quatro partes:

(1) Transformação do diagrama de classes/OCL em uma estrutura interna (lista de objetos), de maneira que cada elemento de entrada seja mapeado para um classe correspondente em formato XMI (do inglês, XML *Metadata Interchange*);

(2) Identificação dos parametros nas restrições expressas em OCL através da utilização de gabaritos. Nesta etapa são especificados marcas que visam informar como as instâncias do modelo devem ser tratadas;

(3) Geração de valores para os parametros identificados, onde um valor do tipo *string* pode ser válido ou *null*; um valor do tipo numérico ou *date* receberá dados conforme critério de valor limite e particionamento de classes de equivalencia (ou aleatóios caso não haja restrição identificada na expressão OCL); e valores do tipo *boleanos* receberão um valor verdadeiro e outro valor falso;

(4) Geração das instancias de teste com base no produto cartesiano dos valores gerados na etapa anterior.

As instâncias geradas pelos método proposto pelos autores permite que os casos de teste sejam executados na ferramenta USE para que o analista possa validar visualmente se o comportamento das instâncias de teste estão condizentes com o que foi especificado na análise dos requisitos. Dessa maneira, o resultado esperado para uma execução de instância de teste é de conhecimento apenas do analista.

2.7.1 Oportunidades de pesquisa

Através da leitura dos artigos selecionados foi possível encontrar oportunidades de pesquisa, dentre as quais algumas estão diretamente relacionadas a este trabalho. Essas oportunidades estão descritas a seguir:

- Nenhum dos trabalhos encontrados durante o mapeamento utiliza técnicas *lightweight* de validação do modelo conceitual dentro do escopo da TBM para geração de casos de teste e oráculos de teste para um sistema sob teste. Há uma divisão muito clara entre os testes gerados para implementação e os

testes gerados para uma especificação. A junção dessas duas abordagens representa uma oportunidade de pesquisa.

- A maioria dos trabalhos relacionados utilizam OCL de uma maneira mais simplificada – apenas para restrições simples sobre valores de atributos – não havendo estratégias definidas no caso de restrições sobre as multiplicidades de um modelo.
- Outro fator observado está relacionado a utilização de mais de um diagrama da UML para geração dos casos de teste, onde os mais utilizados são os diagramas de sequencia, maquina de estados, atividades e classes. Poucas pesquisas utilizam apenas o diagrama de classes e OCL para geração dos testes.
- A utilização da ferramenta USE para geração de testes, seja através da validação ou da animação de um modelo conceitual ainda é pouco explorada.
- Existem poucas investigações empíricas que visam entender o impacto relacionado ao processo de geração automático de testes a partir da UML dentro do ciclo de desenvolvimento do software.
- Desenvolvimento de novas estratégias para seleção dos casos de teste a serem utilizados quando ocorrer crescimento exponencial do número de possibilidades a serem geradas, de modo a diminuir as redundâncias geradas.

3 Avaliação quase-experimental sobre a aplicação da Hipótese do Escopo Reduzido

Este capítulo descreve uma avaliação quase-experimental sobre a aplicação da hipótese do escopo reduzido na abordagem proposta no trabalho. A seção 3.1 indica o tipo de pesquisa utilizada. A seção 3.2 apresenta o escopo da pesquisa. Na seção 3.3 é apresentado o planejamento. A seção 3.4 demonstra a operacionalização e a seção 3.5 apresenta os dados obtidos.

3.1 Tipo de Pesquisa

Apesar do presente trabalho apresentar uma abordagem mais técnica, foi necessária a realização de um método empírico que dê evidências de que a presente proposta é viável. Nesse contexto, optou-se pela realização de um *quase*-experimento. Os quase-experimentos são investigações empíricas semelhantes aos experimentos, no entanto, o tratamento das amostras não é realizada de maneira aleatória, onde estas são obtidas por conveniência. O método é realizado seguindo os passos propostos por Wohlin, Höst e Regnell (2012).

As etapas do processo *quase*-experimental são dispostas na seguinte ordem: definição do escopo, planejamento, operação, análise interpretativa e apresentação.

3.2 Escopo do *Quase*-Experimento

Durante a etapa da definição do escopo são definidos os objetivos e os objetos de estudo. Para tal é aplicada a abordagem GQM (do inglês, *Goal, Question, Metric*) proposta em (BASILI, CALDIERA e ROMBACH, 1994). O GQM é dividido hierarquicamente em 3 camadas: nível conceitual (objetivos), nível operacional (questões) e nível quantitativo (métricas).

Tabela 3: Template GQM abordado

Objetivos	Objeto	Propósito	Foco	Perspectiva	Contexto
	Validação do modelo	Avaliar	Tempo	Pesquisador	Ambiente de experimentação
Questões:	Qual o tamanho do modelo factível de se operar na prática?				

	Considerando a Hipótese do Escopo Reduzido, qual é o número de instâncias de cada tipo ideal para execução do método no ambiente de experimentação?
Métricas	O tamanho do modelo, a quantidade de *Snapshots* e o tempo gasto para geração deles.

3.3 Planejamento

A fase de planejamento é onde o contexto da experiência é detalhado. São apresentados aspectos do seu ambiente de experimentação, as variáveis de entrada e saída são identificadas, as hipóteses são declaradas de maneira formal e é escolhido o desenho experimental a ser utilizado.

3.3.1 Ambiente de Experimentação

O processo de experimentação foi realizado em laboratório utilizando uma máquina com a seguinte configuração: Sistema Operacional *Windows* 10 x64, processador Intel i5 4440 com 3,1 Ghz, 8 *gigabytes* de memória RAM, Eclipse Jee Neon 1.1, USE 4.2.0 (modificada).

O tempo de execução dos testes foi limitado em 7200 segundos. O tempo foi limitado a este número em razão de possíveis travamentos da ferramenta USE, que ocorreram em testes prévios após esse período de tempo.

3.3.2 Identificação das variáveis

As variáveis podem ser descritas como dependentes (saída) e independentes (entrada). As variáveis de saída representam os dados a serem observados após o tratamento das variáveis independentes. No contexto deste trabalho a variável dependente é a quantidade de *snapshot* gerado enquanto que a variável que sofre o tratamento é o número de objetos por classe (*m*) possíveis de acordo com a hipótese do escopo reduzido.

3.3.3 Hipóteses da Pesquisa

A utilização de um método como a hipótese do escopo reduzido aliada à aplicação de validação do modelo através do ASSL sugere que o número de *snapshots* cresça

exponencialmente à medida que o número de objetos participantes aumenta. Assim, foi possível definir as seguintes hipóteses:

- Hipótese H_0: o tempo gasto para geração dos casos de teste não tem influência significativa utilizando *m* = 2 e *m* = 3. Onde *m* é igual ao número de objetos de cada tipo de classe;
- Hipótese H_1: o tempo gasto para geração dos casos de teste explode exponencialmente à medida que o número de objetos do modelo cresce.

3.3.4 Desenho Experimental

O desenho experimental escolhido para esse teste foi o de pós-teste sem grupo de controle. Nesse caso, cada uma das amostras foi submetida a dois tratamentos diferentes a fim de que se possa comprar o resultado ao final de cada teste.

3.4 Operacionalização

O *quase*-experimento consistiu na execução e monitoramento da geração de *snapshots* para um modelo conceitual simples e fictício, que foi divido em seis módulos. A figura 15 apresenta o modelo testado.

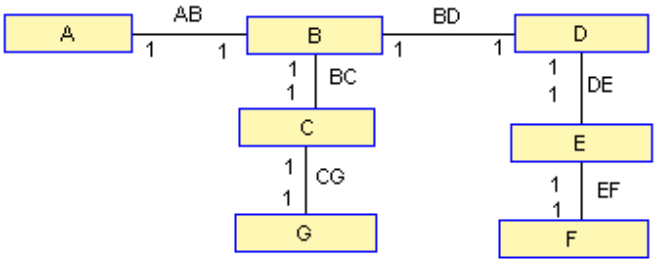


Figura 15: Modelo ficticio para teste da hipótese H_1

Por se tratar de um *quase*-experimento, o modelo foi dividido por conveniência seguindo uma lógica de dependência entre os relacionamentos. A tabela 4 apresenta os dados obtidos na fase de pré-teste.

Tabela 4: Divisão dos módulos do modelo a ser testado

Módulo	Classes Participantes
1	A,B
2	A,B,C
3	A,B,C,D
4	A,B,C,D,E
5	A,B,C,D,E,F
6	A,B,C,D,E,F,G

Cada um dos módulos recebeu tratamento com *m* = 2 e *m* = 3. No modelo de exemplo contendo as classes *A*, *B*, *C*, *D*, *F* e *G* existem 6 relacionamentos bidirecionais com restrições de multiplicidade de um para um. É necessário que o analista especifique ambas as direções como invariantes que serão carregadas dinâmicamente. Dessa forma, o modelo com 6 relacionamenteos terá na verdade 12 restrições de multiplicidades especificadas como invariantes OCL.

3.5 Análise e Interpretação

A escolha do método utilizando a hipótese do escopo reduzido com até dois objetos de cada classe é em razão da enorme quantidade de cenários redundantes gerados e do tempo gasto que cresce exponencialmente à medida que o número de classes e relacionamentos aumenta. O limite inferior de casos gerados com até dois objetos de cada tipo para um modelo com n classes, no pior caso, considerando todas as classes interligadas por relacionamentos, é dado por:

$$N = 2^{c(2n,\,2)-n}$$

Onde $2n$ é o número máximo de objetos possíveis e $C(2n, 2) - n$ é o número de combinações de relacionamentos possíveis entre esses objetos menos os *links* de objetos do mesmo tipo.

É possível observar que há uma explosão exponencial do número de *snapshots* gerados à medida que o número de relacionamentos aumenta. A utilização de até três objetos de cada tipo aumentaria exponencialmente o número de combinações possíveis. A

tabela 5 apresenta os dados referentes a um experimento realizado com dois e três objetos em um modelo conceitual.

Tabela 5: Resultado obtido com até dois e até três objetos

Configuração	*Snapshots* (*m* = 2) Limite Inferior	Tempo *m* = 2	*Snapshots* (*m* =3) Limite Inferior	Tempo *m* = 3
6 relacionamentos 7 classes	16.777.216 (estimado)	>7200s	68.719.476.736 (estimado)	>7200s
5 relacionamentos 6 classes	1.048.576	1560s	1.073.741.824 (estimado)	>7200s
4 relacionamentos 5 classes	65.536	50s	16.777.216 (estimado)	>7200s
3 relacionamentos 4 classes	4.096	12s	262.144 (estimado)	>7200s
2 relacionamentos 3 classes	256	4s	40.096	41s
1 relacionamento 2 classes	16	1s	64	2s

O *quase*-experimento limitou o tempo total máximo para geração dos cenários em 2 horas. No entanto, uma configuração com 6 relacionamentos e 7 classes com até 3 objetos de cada tipo poderia levar inúmeros dias para que fosse completada. É importante observar que os tempos obtidos durante o *quase*-experimento podem variar conforme o ambiente utilizado para experimentação. É possível que dependendo do número de classes e

relacionamentos, em máquinas mais potentes, possa ser utilizado a hipótese do escopo reduzido com até 3 objetos de cada tipo.

4 Abordagem Proposta

Este capítulo descreve a abordagem proposta para geração de casos de teste abstratos a partir de diagramas UML enriquecidos com restrições OCL. A seção 4.1 apresenta uma visão geral da abordagem proposta. A seção 4.2 demonstra a proposta abordada com um exemplo de aplicação.

4.1 Visão Geral

A abordagem proposta neste trabalho é baseada nos conceitos empregados em TBM, onde o processo de teste de *software* é realizado por intermédio da construção de modelos de teste que representam o estado estrutural e comportamental de um SST. Deste modo, o objetivo geral deste trabalho trata da geração semiautomática de casos de teste a partir de modelos de classe UML, enriquecidos com restrições em OCL, de maneira a contemplar uma cobertura satisfatória para testes funcionais. A abordagem tem como foco usuários analistas de requisitos que compreendem e especificam seus modelos corretamente utilizando UML, OCL e ASSL..

Para atingir tal objetivo é proposto um processo sistematizado compreendendo duas abordagens, uma para geração dos *snapshots* de teste e outra para seleção dos *snapshots* gerados. A geração dos casos de teste compreende três artefatos principais: a própria especificação conceitual do domínio, *scripts* ASSL para manipulação das instâncias do modelo e arquivos contendo invariantes OCL, que devem ser carregadas dinamicamente.

No contexto deste trabalho foi definido um caso de teste como uma dupla (*estado*, *situação*), onde o *estado* é um conjunto de objetos e associações do modelo conceitual (*snapshot)* e a *situação* representa a avaliação desse estado. Um *estado* é avaliado como válido quando não viola nenhuma das restrições do modelo e inválido no caso contrário. A figura 16 representa um ponto de vista geral da abordagem proposta.

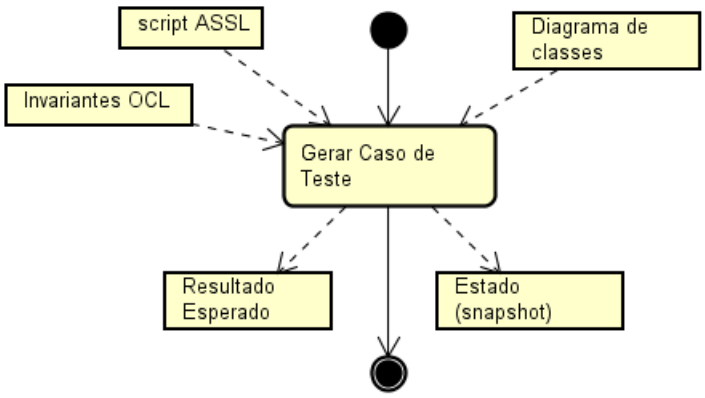


Figura 16: Visão geral do processo de geração dos casos de teste

O princípio básico dos métodos para geração de casos de teste consiste na criação manual de *scripts* ASSL a partir do modelo de classes e de invariantes OCL que informem as restrições funcionais do sistema sob teste.

Cada *script* ASSL contém uma sequência de instruções responsável por gerar sequencialmente todos os estados do sistema com até dois objetos de cada classe (hipótese do escopo reduzido). A validação do estado é feita pela ferramenta USE que avalia se todas as restrições foram atendidas e quando uma ou mais restrições são violadas, o *snapshot* gerado pelo *script* ASSL é inválido. A ferramenta USE atua como um mecanismo provedor de um oráculo na geração dos casos de teste, isto é, dado um modelo conceitual bem formado, que represente fielmente tanto a estrutura como o comportamento do sistema, a ferramenta USE avalia a consistência do estado gerado, fornecendo o seu resultado esperado.

A seleção das instâncias do modelo a serem utilizadas como casos de teste segue um algoritmo que gera um conjunto de casos de teste que exercitem todas as combinações possíveis, com até dois objetos e que tenham a característica de cobrir todas as situações possíveis (dentro do escopo reduzido) de violação dos invariantes. Por exemplo, se houver 3 invariantes, serão necessários gerar no máximo 2^3 situações de violação dos invariantes. A figura 17 apresenta uma visão mais detalhada da relação entre os elementos da abordagem.

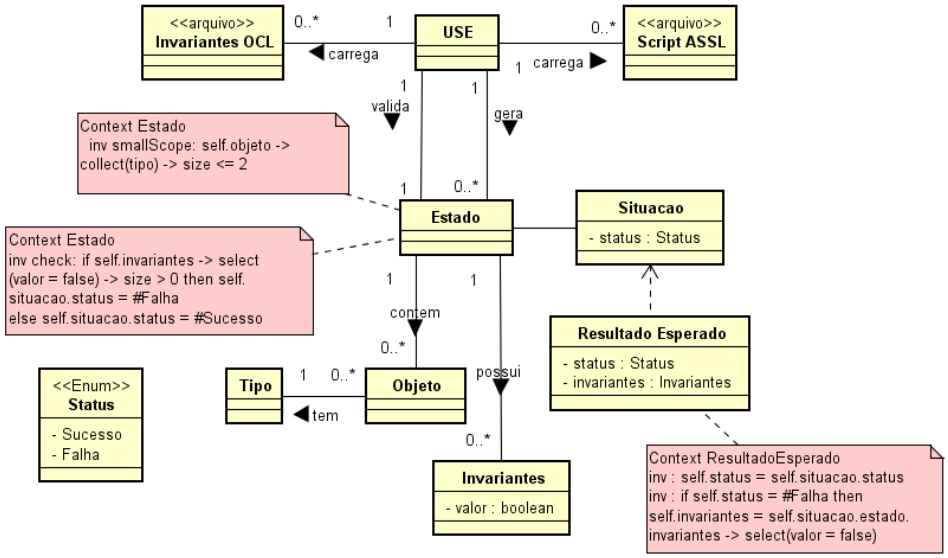


Figura 17: Visão mais detalhada das fases do processo

Cada iteração efetuada pela ferramenta USE durante a execução de um *script* ASSL corresponde a um caso de teste possível. Esse caso de teste é gerado com relação ao que foi especificado pelo analista no *script* ASSL. A execução do *script* ASSL pode inviabilizar a geração dos casos de teste, pois existe a possibilidade de que a quantidade de *snapshots* gerados cresça exponencialmente. Este trabalho propõe duas abordagens que objetivam minimizar o problema do crescimento exponencial do número de *snapshots*, a primeira relacionada à maneira como os casos de teste devem ser criados e a segunda diz respeito ao processo de seleção dos casos obtidos durante a execução da validação do modelo. A figura 18 apresenta o fluxo das atividades da abordagem para geração e seleção dos cenários que serão utilizados como casos de teste.

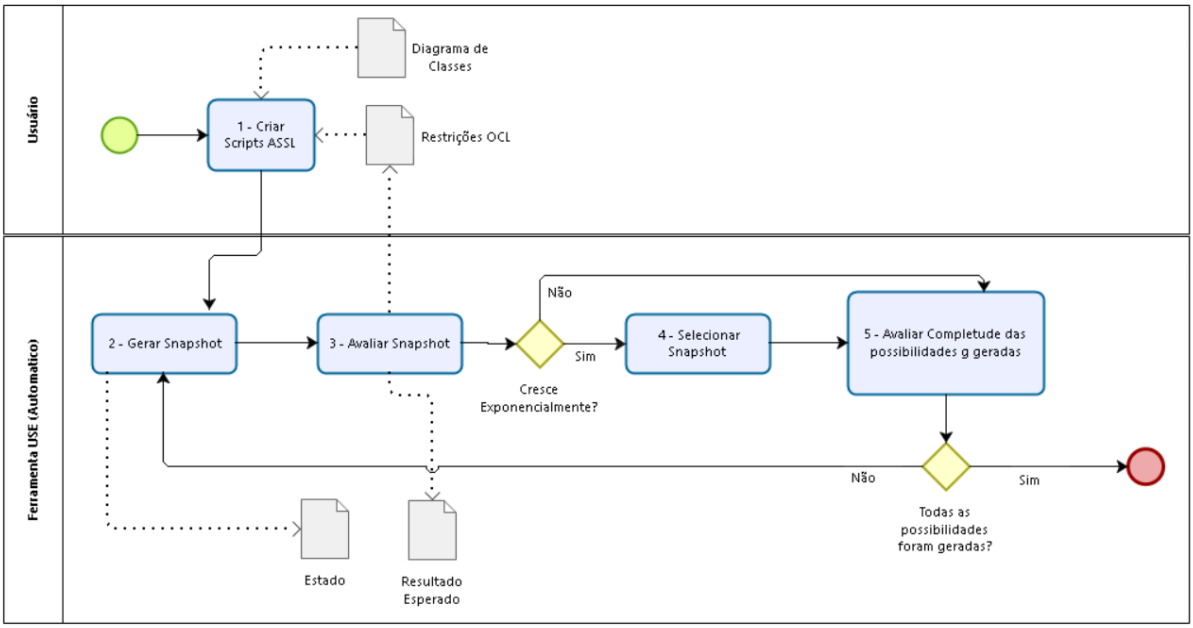


Figura 18: Fluxo das atividades da abordagem proposta

A primeira atividade do processo de geração dos casos de teste consiste na geração dos *scripts* ASSL com base no modelo conceitual, que represente o sistema sob teste e suas restrições através de invariantes OCL. Em seguida, esses *scripts* ASSL são executados através da ferramenta USE, que irá gerar de maneira sequencial todos os *snapshots* que representam instâncias do modelo conceitual em um determinado instante do tempo. Cada *snapshot* gerado é avaliado com relação à consistência de suas invariantes pela ferramenta USE. Em tempo de execução, primeiro existe uma verificação para saber se o número de *snapshots* possíveis é muito grande, nesse caso há a seleção dos casos que serão utilizados ao final do processo. A seleção dos casos é interrompida quando todas as possibilidades de violação de invariantes forem alcançadas.

4.2 Exemplo de utilização

Para exemplificar a aplicação da abordagem, será utilizado o domínio *Royal & Loyal* (WARMER e KLEPPE, 2003) modificado. O modelo criado por Warner e Kleppe descreve um modelo de uma companhia fictícia, que opera com cartões de fidelidade, de maneira que quando um cliente faz uma compra com o cartão, ele ganha pontos referentes ao valor dessa compra. Tais pontos podem ser utilizados para descontar os custos de transações futuras. A figura 19 apresenta o diagrama de classes para o domínio *Royal & Loyal*.

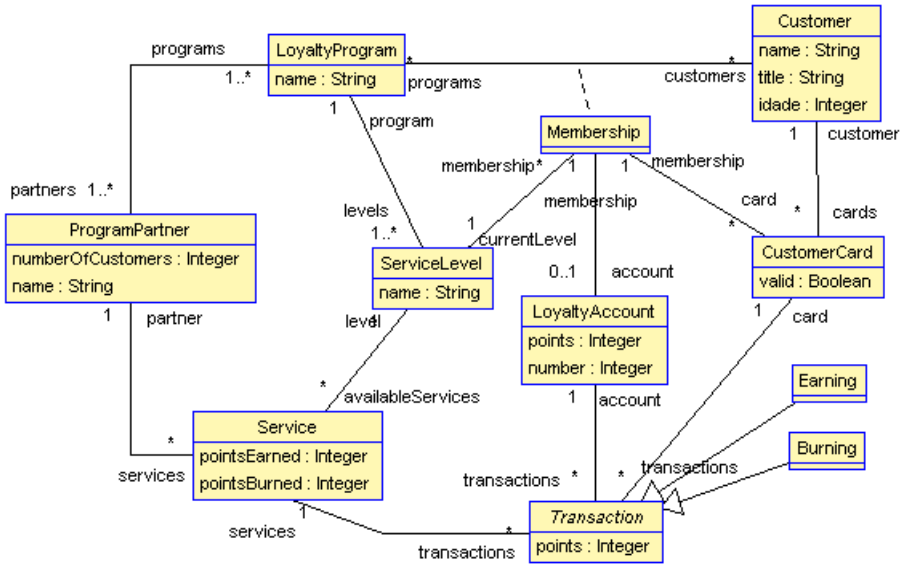


Figura 19: Modelo Royal & Loyal (WARMER & KLEPPE, 2006) modificado.

Cada cliente pode se associar a um programa de fidelidade com um parceiro do programa. O parceiro do programa é a empresa que oferece os pontos em compras. Um cliente pode possuir alguns destes cartões, desde estejam associados a um programa de fidelidade por cartão. Cada transação utilizando o cartão de cliente é gravado em uma conta de fidelidade, que registra o número de pontos ganhos e utilizados. Os pontos gastos são aqueles usados para fazer uma compra. O número de pontos ganhos e usados por uma transação é registrada na classe de serviço.

Cada cliente de cada parceiro de programa pode entrar no programa de fidelidade preenchendo um formulário para obter um cartão de membro. Os objetos da classe *Cliente* representam pessoas que entraram no programa. O cartão de membro, representado pela classe *CustomerCard*, é emitido para uma pessoa. A maioria dos programas de fidelidade permite aos clientes economizar pontos de bônus. Cada parceiro do programa individual decide quando e quantos pontos de bônus são atribuídos para uma determinada compra. Os pontos de bônus salvos podem ser usados para "comprar" serviços específicos de um dos parceiros do programa. Para contabilizar os pontos de bônus que são salvos pelo cliente, cada associação pode ser associada a uma conta de fidelidade.

Neste cenário, existem dois tipos de transações. Em primeiro lugar, existem transações em que o cliente obtém pontos de bônus. Segundo, há transações onde o cliente consome pontos de bônus. As estações de serviço oferecem descontos aos clientes, mas não

oferecem e nem aceitam pontos de bônus e, por conseguinte, não efetuam transações na fidelidade.

Além das regras expressas no modelo em formas de cardinalidade, também estão presentes as seguintes regras de negócio:

- O número de cartões válidos para cada cliente deve ser igual ao número de programas de fidelidade do qual ele participa;
- Quando um Programa de Fidelidade não ganha e nem gasta pontos, seus membros não possuem uma Conta de Fidelidade;
- O número de clientes de um Parceiro do Programa de Fidelidade deve ser igual ao número de clientes que estão associados a um programa de fidelidade oferecido por este parceiro;
- O limite máximo de pontos que podem ser queimados por cada parceiro de programa é 10 mil;
- O número de clientes de um Programa Parceiro dever ser maior ou igual a zero;
- O número de caracteres permitidos para todos os valores do tipo *string* deve ser maior que zero e menor que 25;
- O número de pontos ganhos e gastos por um serviço deve ser igual ou maior do que zero;
- Um Cliente deve ter mais de 18 anos.

A primeira fase da criação dos casos de teste diz respeito a maneira como os *scripts* ASSL devem ser especificados, para isso, é necessário aplicar um método sistematizado que possibilite ao analista uma maneira de modelar o comportamento do sistema.

4.2.1 Abordagem para geração dos Casos de Teste

A abordagem para geração das instâncias é composta por quatro métodos distintos:

- (M1) Método para geração de testes sobre os valores;
- (M2) Método para geração de teste sobre as Regras de Negócio;
- (M3) Método para geração de testes sobre as Operações;
- (M4) Método para geração de testes para a Estrutura do modelo.

Os três primeiros métodos fornecem informações à cerca do comportamento abstrato da aplicação de uma maneira mais generalizada, isto é, cenários de objetos são gerados sem que haja necessariamente uma operação envolvida, provendo meios para a verificação estrutural do sistema em teste. O quarto método (M4), por sua vez, gera casos de teste para as operações do sistema com base nas suas pré e pós-condições.

4.2.2 Método para geração de testes sobre os valores (M1)

Este método trata da geração dos testes com relação aos valores dos atributos de cada classe. Tal procedimento deve obedecer aos seguintes passos:

1. Identificar em cada classe do modelo quais são as restrições sobre os valores dos atributos. E para cada classe que tenha no mínimo uma restrição, especificar um arquivo contendo as regras sobre os valores dos atributos dela em formato de invariantes OCL;
2. Especificar para cada classe do domínio um procedimento ASSL aplicando a técnica de valor-limite e equivalência de classes sobre o valor de cada atributo;
3. Carregar no USE o modelo de classes livre de restrições de multiplicidade;
4. Carregar dinamicamente na ferramenta USE as invariantes contendo as restrições sobre os valores;
5. Executar procedimentos ASSL através da ferramenta USE.

Procedimento 1: Passos para execução de M1

O primeiro passo do algoritmo para geração de teste sobre os valores consiste na identificação das regras impostas aos valores do domínio. Tais regras devem estar

especificadas em um arquivo separado do modelo para que possam ser carregadas dinamicamente. A figura 20 apresenta as regras sobre os valores dos atributos das classes do modelo em OCL.

```
context s:Service
    inv test1: s.description.size > 0 and
               s.description.size < 25
    inv test2: s.pointsBurned >= 0
    inv test3: s.pointsEarned >= 0

context sl:ServiceLevel
    inv test1: sl.name.size > 0 and
               sl.name.size < 25

context c:Customer
    inv test1: c.title.size > 0 and
               c.title.size < 25
    inv test2: c.name.size > 0 and
               c.title.size < 25

context cc:CustomerCard
    inv test1: cc.printedName.size > 0 and
               cc.printedName.size < 25

context la:LoyaltyAccount
    inv test1: la.points >= 0

context t:Transaction
    inv test1: t.points >= 0

context pp:ProgramPartner
    inv test1: pp.numberOfCustomers >= 0

context lp:LoyaltyProgram
    inv test1: lp.name.size > 0 and
               lp.name.size < 25
```

Figura 20: Regras sobre os valores dos atributos

No modelo da *Royal & Loyal* (Figura 19) são identificadas 8 classes contendo valores sujeitos a esse tipo de teste. Após a identificação dos atributos cujos valores apresentam algum tipo de regra, deve-se então criar um procedimento ASSL para cada classe identificada. Tal procedimento, conforme já mencionado, deve conter valores baseados em critérios de teste de particionamento em classes de equivalência e valor-limite. A figura 21 apresenta um exemplo de procedimento ASSL para a classe *Service*.

```
procedure ServiceUnitTests()
var
   service : Service;
begin
    service := Create(Service);
    [service].description := Try([Sequence{'','a', '123456789', '1234567890'}]);
    [service].pointsBurned := Try([Sequence{-1, 0, 1}]);
    [service].pointsEarned := Try([Sequence{-1, 0,1}]);
end;
```

Figura 21: Exemplo de procedimento ASSL para teste da classe Service

As instruções indicam que a classe testada é a classe *Service* e que para cada atributo identificado serão geradas todas as possibilidades de valores válidos e não válidos seguindo o particionamento de classes e a análise do valor-limite, por intermédio da instrução *Try*.

O próximo passo do método diz respeito ao carregamento do arquivo de diagrama de classes livre de restrições de estrutura. Após isso, é necessário carregar o arquivo contendo as invariantes OCL e executar o procedimento ASSL para que o modelo seja dinamicamente animado pela ferramenta USE. A figura 22 apresenta os comandos utilizados para execução do modelo.

```
constraints -load unitConstraints.invs
gen start -s -nb -dc -ac royaltests.assl genUnitTestService()
```

Figura 22: Comandos para execução do modelo

Durante a execução dos procedimentos, a ferramenta USE irá interpretar as instruções, animar o modelo e validar o resultado de cada iteração com relação às invariantes dinamicamente carregadas. Após a execução de todos os procedimentos necessários, são gerados casos de teste independentes de plataforma, utilizando os objetos e seus respectivos valores de entrada e resultado esperado.

Como o espaço de busca para esse modelo de teste é pequeno, então não há necessidade de reduzir o número de casos encontrados, desta maneira, todos os casos de teste gerados são utilizados.

4.2.3 Método para geração de testes sobre regras de negócio (M2)

Regras de negócio, neste trabalho, dizem respeito às regras que não são apenas sobre valores simples e nem exclusivamente sobre relacionamentos. Geralmente, esses tipos de regras envolvem ambos os tipos de teste ao mesmo tempo, de modo a contemplar a estrutura e os valores dos atributos. Para esse tipo de regra, é necessário seguir o procedimento:

1. Identificar cada regra de negócio presente no modelo;
2. Para cada regra de negócio identificada, especificar procedimento ASSL que exercite à regra de negócio. Cada procedimento ASSL deve estar em conformidade com o critério de seleção de testes de equivalência de classes, valor-limite e hipótese do escopo reduzido quando necessário;
3. Carregar no USE o modelo de classes livre de restrições de multiplicidade;
4. Carregar dinamicamente na ferramenta USE as invariantes contendo as restrições OCL;
5. Executar procedimentos ASSL através da ferramenta USE.

Procedimento 2: Passos para execução de M2

A primeira etapa do processo de geração dos cenários de teste para regras de negócio está ligada a identificação das mesmas no modelo. A figura 23 apresenta um conjunto de regras de negócio identificadas para o domínio *Royal & Loyal*.

```
context c: Customer
    inv test2:c.programs -> size = cards ->
            select(valid = true) -> size

context lp:LoyaltyProgram
    inv test: lp.partners.services ->
            forAll(pointsEarned = 0 and pointsBurned = 0)
            implies lp.membership -> select(account -> size > 0) ->
            isEmpty and lp.partners.services ->
            forAll(pointsEarned > 0 or pointsBurned > 0) implies
            lp.membership -> select(account -> size >0) -> notEmpty

context pp:ProgramPartner
    inv test: pp.numberOfCustomers = pp.programs.customers -> asSet -> size

context lp:LoyaltyProgram
    inv test: lp.partners.services.transactions ->
            select(oclIsKindOf(Burning)) -> collect(points) -> sum < 10000
```

Figura 23: Conjunto de Regras de Negócio

Com base nas invariantes identificadas é necessário especificar o modelo de teste através do ASSL de modo a permitir a manipulação da ferramenta USE. Cada invariante OCL é transformada em um procedimento ASSL responsável por dizer ao USE como o modelo deverá se comportar durante sua execução. Através dos valores expressos em OCL é possível identificar limites de fronteira e classes de equivalência, permitindo que este tipo de informação seja também expressada no ASSL. A figura 24 apresenta um fragmento de um procedimento para geração de teste para regras de negócio.

```
procedure genSmallScope(loyaltyProgram: Integer, customerCard: Integer)
var
    loyaltyPrograms : Sequence(LoyaltyProgram),
    customer : Customer,
    customerCards: Sequence(CustomerCard);
begin

    customer:= Create(Customer);

    if[loyaltyProgram > 0] then
        begin
            loyaltyPrograms := CreateN(LoyaltyProgram, [loyaltyProgram]);
        end;
    if[customerCard > 0] then
        begin
            customerCards := CreateN(CustomerCard, [customerCard]);

            for cc: CustomerCard in [CustomerCard.allInstances -> asSequence]
                begin
                    [cc].valid := Try([Sequence{true, false}]);
                end;

        end;

    if[loyaltyProgram > 0] then
        begin
            Try(Membership, [loyaltyPrograms],[Customer.allInstances -> asSequence]);
        end;

    if[customerCard > 0] then
        begin
            Try(CustomerCCard, [Customer.allInstances -> asSequence], [customerCards]);
        end;


end;
```

Figura 24: Procedimento para manipulação do modelo para uma Regra de Negócio

O procedimento da figura 24 diz respeito à regra de negócio no qual um cliente deve possuir um número de cartões válidos de acordo com o número de programas que ele é membro. Novamente, os parâmetros inteiros da assinatura do procedimento ASSL estão ligados a geração do escopo reduzido, sendo necessário gerar para esses parâmetros todas as combinações possíveis para zero, um ou dois objetos de cada tipo.

4.2.4 Método para geração de testes sobre operações (M3)

Operações em um modelo orientado a objetos dizem respeito ao comportamento cuja instância do objeto é capaz de realizar. São partes fundamentais do modelo comportamental do domínio e por isso a criação de testes para operações é uma etapa fundamental do teste de *software*. O método M3 para geração de casos de teste neste trabalho engloba as operações que envolvem a criação e remoção de objetos e deve seguir este procedimento:

1. Transformar todas as pré e pós-operações de cada operação em invariantes OCL;
2. Para cada invariante de pré-condição, adicionar o prefixo "*pre_*";
3. Para cada invariante de pós-condição, adicionar o prefixo "*post_*".
4. Para cada operação do modelo, especificar procedimentos ASSL contendo um cenário de pré-condição válido. No caso de teste sobre a pré-condição, então especificar um cenário de pré-condição inválido. No primeiro caso serão gerados tanto testes positivos quanto negativos, já no segundo caso serão gerados apenas casos negativos;
5. Para cada operação do modelo especificar o comportamento da operação através de procedimento ASSL seguindo os critérios de análise do valor limite, particionamento em classes de equivalência e escopo reduzido;
6. Carregar dinamicamente na ferramenta USE as invariantes contendo as restrições OCL;
7. Executar procedimentos ASSL através da ferramenta USE.

Procedimento 3: Passos para execução de M3

Os procedimentos ASSL trabalham apenas diretamente sobre as invariantes, dessa forma o primeiro passo deste método consiste na transformação das pré e pós-condições especificadas no modelo em invariantes. A utilização dos prefixos é necessária para identificação do tipo e o resultado esperado do teste. Se é um teste sobre a pré-condição ou se é um teste sobre a pós-condição. Dessa maneira, para um invariante cujo prefixo “*pre_*” tem valor verdadeiro, toda invariante com prefixo “*post_*” deverá ter um resultado verdadeiro para casos de teste positivos e um resultado falso para casos de teste negativos. Quando o teste for sobre a pré-condição, esta deverá ter valor falso, de modo que todos os cenários gerados para a pós-condição também tenham como resultado o valor falso, ou seja,

todos os cenários gerados cuja pré-condição não estiver satisfeita deverá retornar um erro. A figura 25 apresenta um exemplo de cenário de pré condição para a operação "*Enroll*()".

```
procedure Enroll(customer : Customer)
var programPartner: ProgramPartner,
    loyaltyProgram: LoyaltyProgram,
    serviceLevel: ServiceLevel,
    loyaltyAccount: LoyaltyAccount,
    card : CustomerCard,
    service: Service;

begin
    --pre
    if[LoyaltyProgram.allInstances.customers ->
            exists(customer)] then
        begin
            Delete(customer);
        end;
    else
        begin
            customer := Create(Customer);
            customer.[name] := Try([Sequence{'','a','abcdfdswef', 'adehjklçpo'}]);
            customer.[title] := Try([Sequence{'','a','abcdfdswef', 'adehjklçpo'}]);
            customer.[idade] := Try([Sequence{17,18,19}]);

            programPartner := Create(ProgramPartner);
            loyaltyProgram := Create(LoyaltyProgram);
            serviceLevel := Create(ServiceLevel);
            service := Create(Service);
            loyaltyAccount := Create(LoyaltyAccount);

            Try(Membership, [LoyaltyAccount.allInstances ->
                asSequence],[customer]);

            Insert(CustomerCCard, [customer],[card]);
            Insert(MembershipLAccount, [membership], [loyaltyAccount]);
            Insert(MembershipSLevel, [membership], [serviceLevel]);
            Insert(MembershipCCard, [membership], [card]);
        end;

end;
```

Figura 25: Exemplo de procedure para a operação Enroll

A operação *Enroll*() diz respeito ao registro de um cliente ao programa de fidelidade, desta maneira uma pré-condição encontrada é o fato de que esse cliente não pode estar cadastrado no programa de fidelidade em questão. Outras pré-condições dizem respeito aos valores que os atributos do objeto *Customer* podem obter. A pós-condição para essa operação é que o cliente esteja associado ao programa de fidelidade correspondente ao final da execução do registro. A figura 26 apresenta as pré e pós-condições da operação *Enroll*().

```
context lp:LoyaltyProgram
 inv pre_enroll:
    lp.customers -> excludes(Customer)

 inv post_enroll:
    lp.customers -> includes(Customer)
```

Figura 26: Exemplo de pré e pós condições como invariantes

4.2.5 Método para geração de testes de estrutura (M4)

Este método aborda a geração dos testes com relação às regras de multiplicidade dos relacionamentos do modelo, cuja violação indica uma falha de integridade referencial. Para que este método tenha efeito, deve ser utilizado o seguinte procedimento:

1. Transformar todas as regras de multiplicidade do modelo em invariantes OCL;
2. Identificar o número de relacionamentos do modelo, se maior que cinco então o procedimento de geração dos testes deve ser dividido seguindo conjuntos menores de relacionamentos. Tal divisão deve ser feita com base nas regras de multiplicidade especificadas em OCL, onde cada conjunto de regras deve possuir um número mínimo de relacionamentos dependentes entre si;
3. Para cada conjunto de regras especificadas, criar um *script* ASSL com base nas classes e relacionamentos pertencentes ao conjunto. A especificação do teste em ASSL deve ser realizada seguindo o conceito de hipótese do escopo reduzido, com no mínimo zero e no máximo dois objetos de cada classe;
4. Carregar dinamicamente cada conjunto de regras OCL;
5. Executar o *script* ASSL referente às regras, sequencialmente até que todas as invariantes do modelo completo tenham sido violadas.

Procedimento 4: Passos para execução de M4

O primeiro passo deste método consiste na transcrição das regras expressas em multiplicidade (cardinalidade) para regras expressas em invariantes OCL, de modo a permitir o carregamento dinâmico das mesmas, isso possibilita que subconjuntos de restrições de multiplicidade sejam testadas dinamicamente. Como o modelo *Royal & Loyal* (Figura 19) apresenta mais do que cinco relacionamentos, então é necessário que suas regras relacionais sejam divididas em conjuntos menores, de maneira a viabilizar a geração dos testes em tempo hábil. O limite de 5 relacionamentos é derivado do resultado do *quase*-experimento realizado sobre a aplicação da hipótese do escopo reduzido no contexto deste trabalho (ver capítulo 3). É importante que essa divisão seja feita com base nas dependências relacionais entre as classes. A figura 27 apresenta um exemplo de conjunto de invariantes para o modelo *Royal & Loyal* (Figura 19).

```
context b: Burning
    inv ServiceRuleBurning: b.services -> size =  1
    inv LoyaltyAccountRuleBurning: b.account -> size() = 1
    inv CustomerCardRuleBurning: b.card -> size() = 1

context e: Earning
    inv ServiceRuleEarning: e.services -> size =  1
    inv LoyaltyAccountRuleEarning: e.account -> size() = 1
    inv CustomerCardRuleEarning: e.card -> size() = 1
```

Figura 27: Exemplo de regras de cardinalidade transcritas em invariantes OCL

O segundo passo deste método trata da criação do *script* ASSL para manipulação dos objetos do modelo. Aqui é necessário especificar as instruções com base na hipótese do escopo reduzido, afim de que o número de casos gerados seja menor, porém ainda representativo do sistema. A figura 28 apresenta um exemplo de especificação ASSL para os relacionamentos das classes *Burning* e *Earning*.

```
procedure genSmallScope(burning: Integer, earning: Integer)
var
  burnings: Sequence(Burning),
  earnings: Sequence(Earning);

begin
  if[burning > 0] then
    begin
      burnings := CreateN(Burning, [burning]);
    end;

  if[earning > 0] then
    begin
      earnings := CreateN(Earning, [earning]);
    end;

  if[burning > 0] then
    begin
      Try(ServiceTransaction, [Service.allInstances -> asSequence],[burnings]);
      Try(LAccountTransaction, [LoyaltyAccount.allInstances -> asSequence],[burnings]);
      Try(CCardTransaction, [CustomerCard.allInstances -> asSequence],[burnings]);
    end;

  if[earning > 0] then
    begin
      Try(ServiceTransaction, [Service.allInstances -> asSequence],[earnings]);
      Try(LAccountTransaction, [LoyaltyAccount.allInstances -> asSequence],[earnings]);
      Try(CCardTransaction, [CustomerCard.allInstances -> asSequence],[earnings]);
    end;
end;
```

Figura 28: procedimento ASSL para geração dos cenários para as classes *Burning* e *Earning*

Os parâmetros do procedimento indicam o valor combinatório para geração dos cenários com base no escopo reduzido. Assim, para o exemplo da figura 17 são necessários 3^2 procedimentos para que todos os cenários possíveis sejam gerados com zero, um ou dois

objetos de cada tipo (*Burning* e *Earning*). Desta maneira, apenas os valores dos parâmetros mudam a cada iteração. A figura 29 apresenta um exemplo de *script* para execução do ASSL da figura anterior.

```
constraints -load constraints.invs
gen start -s -dc -nb -ac royalTests.assl genSmallScope(0,1)
gen start -s -dc -nb -ac royalTests.assl genSmallScope(1,0)
gen start -s -dc -nb -ac royalTests.assl genSmallScope(1,1)
gen start -s -dc -nb -ac royalTests.assl genSmallScope(0,2)
gen start -s -dc -nb -ac royalTests.assl genSmallScope(1,2)
gen start -s -dc -nb -ac royalTests.assl genSmallScope(2,0)
gen start -s -dc -nb -ac royalTests.assl genSmallScope(2,1)
gen start -s -dc -nb -ac royalTests.assl genSmallScope(2,2)
```

Figura 29: script para execução do ASSL

Os *scripts* são executados até que todas as invariantes tenham sido encontradas seguindo a técnica para seleção dos casos de teste.

4.2.6 Abordagem para seleção dos Casos de Teste

O procedimento de geração de cenários utilizando procedimentos ASSL juntamente hipótese do escopo reduzido com até 3 possibilidades abre margem para que uma enorme quantidade de casos de teste seja gerada. Assim, torna-se necessário algum tipo de seleção para identificar um conjunto satisfatório de cenários. Essa seleção é feita apenas quando o número possibilidades de *snapshots* crescer exponencialmente, podendo ser aplicada a qualquer um dos métodos propostos nas seções anteriores. Neste trabalho a seleção foi aplicada apenas a M4, por se tratar do método com maior possibilidade de crescimento exponencial dos *snapshots* gerados. A abordagem proposta tem como objetivo a seleção de cenários com base nas violações de invariantes, sendo um procedimento baseado nos casos de teste negativos.

1. Identificar número total de invariantes a serem testadas;
2. Gerar todas as combinações possíveis de violação com zero, uma e duas invariantes de cada tipo (hipótese do escopo reduzido). Se um modelo apresenta 6 invariantes, então o número máximo de combinações possíveis para as violações é de 3^6.

Procedimento 5: Procedimento para seleção dos *Snapshots*

A abordagem proposta permite isolar um determinado tipo de falha através da seleção de casos que violem apenas um invariante de cada vez, como também é possível isolar conjuntos dependentes de falhas. Assim, é possível não só diminuir o número total de casos gerado, como também gerar todos os tipos (e combinações) de falhas possíveis. A tabela 6 exemplifica a abordagem empregada com até dois objetos (*m*) de cada tipo de classe (correspondente a um invariante).

Tabela 6: Exemplo de seleção dos casos com base na violação de invariantes

Caso	Invariante A	Invariante B
1	*m=0*	*m=0*
2	*m=0*	*m=1*
3	*m=0*	*m=2*
4	*m=1*	*m=0*
5	*m=1*	*m=1*
6	*m=1*	*m=2*
7	*m=2*	*m=0*
8	*m=2*	*m=1*
9	*m=2*	*m=2*

O procedimento irá armazenar uma tabela para zero, um e dois objetos, seguindo novamente o paradigma da hipótese do escopo reduzido. Desta maneira, no caso da seleção de cenários para dois relacionamentos serão escolhidas 3^2 combinações de invariantes violadas. Durante a busca por situações que violem os invariantes, é possível armazenar os casos positivos encontrados até que a busca pelas violações esteja completa.

4.2.7 Geração dos Casos de Teste

Todos os *snapshots* selecionados são armazenados pela ferramenta USE em uma estrutura interna contendo uma lista de objetos. Cada linha da lista contém uma dupla contendo <*estado*, *situação*>. Onde o *estado* representa o *snapshot* composto por objetos, atributos e *links*, e a *situação* representa a consistência desse *estado*. Em seguida essa lista é transformada em casos de teste independentes de plataforma ao serem traduzidos para formatos como SOIL ou XML. A figura 19 apresenta um exemplo de caso de teste abstrato gerado.

```
-- Script generated by USE 4.2.0

-- Input ->
!new Service
!new Service
!new ProgramPartner
!new LoyaltyProgram
!insert (ProgramPartner1,Service1) into PPartnerService
!insert (ProgramPartner1,Service2) into PPartnerService
!insert (LoyaltyProgram1,ProgramPartner1) into LProgramPartner

-- Expected Result -> Invalid state
-- Error inv -> OneLoyaltyAccOneManyServiceLeves
```

Figura 30:Exemplo de Caso de Teste Gerado em SOIL

O processo de transformação entre os modelos segue os princípios da MDA, onde a estrutura interna (objetos, *links*, atributos e situação) é traduzida para o modelo independente de plataforma (dados de entrada, procedimentos e resultado esperado).

5 Discussão

Este capítulo apresenta uma discussão sobre os resultados obtidos após a execução dos métodos propostos no exemplo especificado no capítulo anterior. A seção 5.1 apresenta dos resultados obtidos com a aplicação da abordagem. A seção 5.2 mostra uma discussão comparativa da abordagem apresentada neste trabalho àquelas descritas em trabalhos correlatos.

5.1 Análise dos resultados obtidos no Exemplo

Os resultados obtidos através do exemplo de utilização realizado sobre o domínio *Royal & Loyal* apresentam indícios de que a geração de casos de teste a partir de técnicas *lightweight* de validação do modelo conceitual é viável. As tabelas 7 e 8 apresentam as métricas do domínio *Royal & Loyal* consideradas para este trabalho.

Tabela 7: Métricas do modelo *Royal & Loyal*

Classes	Atributos testados	Relacionamentos	Operações testadas	Módulos divididos	Restrições de Multiplicidade
11	12	12	3	4	17

Tabela 8: Módulos definidos para o modelo *Royal & loyal*

Módulo (subconjunto de restrições)	Classes	Relacionamentos
1	*Burning, Earning, Service, LoyaltyAccount, CustomerCard*	*serviceTransaction, lAccountTransaction, cCardTransaction*
2	*Service, ProgramPartner, LoyaltyProgram, ServiceLevel*	*lProgramPartner, pPartnerService, serviceSLevel, lProgramSLevel*
3	*LoyaltyProgram, Customers, CustomerCard*	*customerCCard, membership, membershipCCard*
4	*LoyaltyProgram, ServiceLevel,*	*membershipCCard, membershipSLevel,*

	LoyaltyAccount, CustomerCard, Customer	*membershipLAccount*

A tabela 9 apresenta os dados obtidos após experimentos sobre o método aplicado a geração de casos de teste para os valores dos atributos.

Tabela 9: Informações obtidas após execução de M1

	Resultado			
Classe	Sucesso (Casos Positivos)	Falha (Casos Negativos)	Escolhidos	Tempo Gasto Aproximado
Service	8	28	36	3s
ServiceLevel	2	2	4	1s
Customer	6	10	16	1s
CustomerCard	2	2	4	1s
LoyaltyAccount	1	1	2	1s
Transaction	4	5	9	1s
ProgramPartner	2	1	3	1s
LoyaltyProgram	2	2	4	1s

A tabela 10 apresenta os dados referentes aos experimentos sobre o método para geração de cenários de teste para as regras do negócio.

Tabela 10: Informações obtidas após execução de M2

	Resultado			
Módulo	Sucesso (Casos Positivos)	Falha (Casos Negativos)	Escolhidos	Tempo Gasto Aproximado
1	22	42	64	4s
2	5	3	8	1s
3	4	12	16	2s
4	3	6	9	1s
5	3	6	9	1s

A tabela 11 apresenta os dados das simulações para o método sobre a geração de cenários de casos de teste para operações.

Tabela 11: Informações obtidas após execução de M3

Operação	Sucesso (Casos Positivos)	Falha (Casos Negativos)	Escolhidos	Tempo Gasto Aproximado
enroll()	6	26	32	2s
unroll()	1	3	4	1s
generateCards()	1	3	4	1s

A tabela 12 apresenta o resultado dos experimentos sobre a geração de cenários de casos de teste para os relacionamentos.

Tabela 12: Informações obtidas após execução de M4

	Resultado			
Módulo (partição)	Sucesso (Casos Positivos)	Falha (Casos Negativos)	Escolhidos (negativos)	Tempo Gasto Aproximado
1	1.240	536.136	729	83s
2	291	20.155	243	9s
3	15	20	9	3s
4	48	1308	81	5s

Com base na análise dos dados obtidos sobre o domínio *Royal & Loyal* é possível encontrar evidências de que a abordagem proposta é efetiva no que diz respeito à cobertura de testes para o sistema sob teste, uma vez que foram gerados casos de teste para todas as restrições mapeadas. Contudo, apesar da abordagem proposta neste trabalho permitir a seleção de casos de teste para todas as restrições, há uma limitação quanto à qualidade do cenário para o teste, uma vez que essa escolha não é tratada com relação ao nível de representatividade do cenário para o domínio. Ou seja, não há garantias de que um caso escolhido seja melhor que outro não selecionado.. Contudo, um cenário escolhido está, de fato, dentro de uma classe (partição) equivalente a outro não selecionado.

Outro aspecto a ser observado é que o ASSL não representa um meio de se fazer validação do modelo conceitual de maneira extremamente rápida, pois seu algoritmo trabalha de maneira enumerativa criando um *snapshot* de cada vez na ferramenta USE, o que torna o processamento mais lento do que em outras abordagens de validação baseadas

em lógica *booleana*. No entanto, a geração dos casos de teste a partir do ASSL pode se mostrar mais rápida do que se tivesse sido realizada manualmente. Utting e Legeard (2007) estimam que o tempo gasto para especificação de cada caso de teste em um modelo conceitual hipotético seria de 10 minutos. Levando em consideração a estimativa apresentada por Utting e Legeard (2007), seriam necessárias 214,3 horas para que fossem criados manualmente todos os casos de teste para o domínio *Royal & Loyal* descrito neste trabalho. Em contrapartida, neste trabalho foram criados apenas 19 *scripts* ASSL, onde cada *script* relacionado ao teste de estrutura do modelo *Royal e Loyal* levou em média de 12 a 15 minutos para ser criado, enquanto outros *scripts* relacionados a testes menores, como sobre valores, por exemplo, levaram menos tempo para serem produzidos, entre 3 a 7 minutos.

É importante ressaltar que a cobertura alcançada pela abordagem não garante efetivamente a eficiência dos métodos, tendo em vista que o processo de criação manual dos *scripts* ASSL ainda é uma tarefa que demanda tempo (em comparação a possíveis estratégias totalmente automatizadas) e é passível de equívocos por parte do analista.

5.2 Comparação com os trabalhos correlatos

A realização de um comparativo sobre os resultados obtidos entre a abordagem proposta nesta dissertação e os trabalhos correlatos não foi possível devido ao fato de que na maioria dos trabalhos relacionados não foram apresentados resultados sobre a execução dos métodos. Dessa forma, optou-se por realizar um breve comparativo sobre como as abordagens foram aplicadas. Nesse aspecto, os trabalhos que apresentam mais semelhança a este são os de Araújo (2009) e Bizerra *et al*. (2012).

Araújo (2009) propõe um método para validar a conformidade dos processos de negócio em relação às regras de negócio, onde a validação do modelo é baseada na animação de um conjunto de cenários executados através da ferramenta USE. Apesar de também utilizar o diagrama de classes anotado com OCL e utilizar o USE como oráculo, o método para geração dos cenários apresentado por Araújo (2009) é fortemente ligado ao diagrama de atividades, diferentemente deste trabalho que faz uso apenas do diagrama de classes. No trabalho de Araújo, o analista também é responsável pela especificação dos *scripts* ASSL que serão utilizados para geração dos cenários de teste. No entanto, a estratégia utilizada por ele é baseada em caminhos chaves de um diagrama de atividades, enquanto que este trabalho

emprega estratégia baseada no número de objetos participantes do cenário, sendo necessário apenas a utilização do diagrama de classes.

O trabalho de Bizerra *et al*. (2012) propõe um método para geração de instâncias de teste com base apenas em diagrama de classes e OCL, porém esse trabalho não especifica qual é a estratégia utilizada para a geração dos cenários com relação às validações de integridade referencial do modelo, limitando-se a descrever as estratégias para geração de valores para atributos e pré e pós-condições sobre as operações do modelo. Além disso, as instâncias geradas por Bizerra *et al*. (2012) são aplicáveis à fase de análise de requisitos do ciclo de vida do *Software*, isto é, os cenários gerados podem ser carregados na ferramenta USE para que o analista de requisitos possa observar através da animação se aquele modelo está ou não consistente. A proposta de Bizerra *et al*. (2012) é apenas para validar modelos, onde o oráculo para os casos de teste é o analista que observa a animação e a valida, ao contrário deste trabalho onde o oráculo de teste é automatizado com base na execução através da ferramenta USE.

Outros trabalhos assemelham-se a este devido ao fato de serem gerados casos de teste abstratos através da utilização de UML e OCL. Porém, a maioria apresenta os diagramas de sequência e de atividades em conjunto com os diagramas de classes, como visto em (NAYAK e SAMANTA, 2010; BOUQUET *et al*., 2008; BAO-LIN *et al*., 2007). Já as abordagens apresentadas por Francisco e Castro (2012) e Chang eLin(2016) são exclusivamente voltadas às operações do modelo, de modo que não há qualquer menção à estrutura da aplicação. O trabalho de Francisco e Castro (2012) emprega uma estratégia baseada na geração de valores aleatórios e está diretamente ligada a fase de implementação do *software*. Nesse caso, o analista de requisitos tem pouca influência sobre como os testes são projetados.

Já na abordagem apresentada por Gogolla *et al*. (2015), os testes são voltados à transformação entre modelos no contexto da MDA, ou seja, os testes são gerados para testar a consistência das transformações entre os modelos. Diferentemente das propostas vistas em Bizerra *et al*. (2012), Araújo (2009) e Gogolla *et al*. (2015), este trabalho utiliza a aplicação das metodologias em um contexto não só de análise como também de implementação, uma vez que os casos de teste gerados podem ser utilizados durante o processo de codificação do *software* para testar o sistema que estiver sendo desenvolvido.

O diferencial deste trabalho em relação aos demais consiste na utilização de um único diagrama da UML para geração dos casos de testes e da geração automática do oráculo de teste a partir da validação *lightweight* da especificação do sistema.

6 Conclusões

Este capítulo apresenta a conclusão da dissertação. A seção 6.1 tece as considerações finais do trabalho. A seção 6.2 apresenta as contribuições produzidas com esta dissertação. Por fim, a seção 6.3 descreve as limitações deste trabalho, bem como aponta sugestões de futuros trabalhos.

6.1 Considerações Finais

Este trabalho apresentou uma abordagem sistematizada para a geração de casos de teste independentes de plataforma. A proposta apresentada segue os princípios dos métodos *lightweight*, usados na verificação de modelos, para a geração dos casos de teste funcionais sobre as restrições do modelo.

Por intermédio da utilização de diagramas de classes anotados com restrições de negócio em OCL é possível gerar casos de teste juntamente com os oráculos de teste a partir da validação da especificação do *software*. Os casos de teste gerados a partir de uma especificação correta do sistema garantem que os requisitos funcionais do sistema serão testados sem os problemas causados pela má interpretação dos requisitos por parte de um analista de teste trabalhando de forma independente.

O analista de teste, muitas vezes, não está tão familiarizado com o domínio do sistema como o analista especializado em modelagem de requisitos do *software*. O problema dos erros de interpretação é minimizado quando as especificações dos testes passam a ser de responsabilidade do analista que especificou o sistema. A abordagem proposta neste trabalho possibilita que os analistas de requisitos que saibam modelar corretamente sistemas utilizando UML e OCL, além da ASSL realizem a especificação dos testes funcionais para um sistema.

6.2 Contribuições

A principal contribuição deste trabalho diz respeito à geração dos casos de teste independentes de plataforma e de seus oráculos de teste a partir da validação *lightweight* da especificação do sistema utilizando ASSL e a hipótese do escopo reduzido através da ferramenta USE. Cada artefato que compõe a abordagem possui uma contribuição especifica. Essas contribuições foram observadas através do exemplo de utilização que deu indícios de que a abordagem é eficaz para geração dos casos de teste.

A primeira contribuição especifica da abordagem são os métodos para geração de cenários utilizando heurísticas baseadas não só na hipótese do escopo reduzido como também em classes de equivalência e valor-limite.

Outra contribuição está relacionada ao procedimento para seleção dos casos de teste quando o número de cenários possíveis cresce de maneira exponencial. Nesse caso é aplicado um procedimento baseado no teste de robustez que seleciona todas as combinações possíveis de violação de invariantes com até dois objetos. Também é visto como contribuição o processo de modularização do teste sobre a estrutura da aplicação, onde um modelo considerado complexo é testado utilizando subconjuntos de restrições de multiplicidade especificadas como invariantes OCL.

6.3 Trabalhos Futuros

A natureza enumerativa do algoritmo utilizado pela ferramenta USE para geração de cenários a partir do ASSL torna lento o processamento de geração dos casos de teste. Logo, foi observado a necessidade de implementação de algum algoritmo que seja mais eficiente do que o algoritmo utilizado pela ferramenta USE para geração de cenários com ASSL. Uma possibilidade estaria na utilização de pacotes de satisfatibilidade lógica (do inglês, *Propositional Satisfiability Problem*, ou SAT) com o *plugin Model Validator* do USE (KUHLMANN, HAMANN e GOGOLLA, 2011).

Outra limitação diz respeito às operações que utilizem a construção OCL “@pre” para operações do modelo, já que o ASSL não apresenta meios para recuperação de informações sobre o estado anterior de um objeto do cenário. Trabalhos futuros podem estar relacionados às operações desse tipo.

Os cenários gerados representam casos de teste abstratos, de modo que os procedimentos específicos para execução dos testes não são gerados. Um trabalho futuro interessante envolve a geração automática dos procedimentos a partir da execução do ASSL durante a geração do *snapshot*.

Finalmente, a abordagem é passível de automatização no que diz respeito à fase de criação dos *scripts* ASSL. Trabalhos futuros podem automatizar a técnica para geração dos *scripts* ASSL tornando a abordagem mais produtiva.

Referências

ARAUJO, M. B. **Um método para validar a conformidade de processos de negócio com regras de negócio**, 2010. Dissertação (Mestrado em Informática) – Programa de Pós-Graduação em Informática – Núcleo de Computação Eletrônica – UFRJ, Rio de Janeiro, Brasil.

BAO-LIN, L; ZHI-SHU, L; QING, L; CHEN, Y. **Test Case automate Generation from UML Sequence diagram and OCL expression**. Internacional Conference on Computational Intelligence and Security. p. 15-19, 2007.

BIOLCHINI, J.; MIAN, P. G.; NATALI, A. C.; TRAVASSOS, G.H. **Systematic Review in *Software* Engineering**: Relevance and Utility. Technical Report. PESC - COPPE/UFRJ. Brazil, 2005, http://www.cos.ufrj.br/uploadfiles/es67905.pdf

BIZERRA, E; SILVEIRA, D S; CRUZ, M; WANDERLEY, F. **A method for generation of tests instances of models from business rules expressed in OCL**. IEEE Latin America Transactions, v. 10, n. 5, p. 2105-2111, 2012.

BOUQUET, F. GRANDPIERRE, C. LEGEARD, B. PEUREX, F. **A Test Generation Solution to Automate *Software* Testing**. Proceeding. AST '08 Proceedings of the 3rd international workshop on Automation of *software* test. p.45-48. 2008

CABOT, J; BRAMBILLA, M; WIMMER, M. **Model-driven *software* engineering in practice**. Synthesis Lectures on *Software* Engineering, v. 1, n. 1, p. 1–182, 2012.

CHANG, C; LIN, N. **A constraint-based framework for test case generation in method-level black-box unit testing**. J. Inf. Sci. Eng. v. 32, n. 2, p. 365-387, 2016.

CUNHA, C A. **Uma abordagem para a transformação de regras de negócio na arquitetura dirigida por modelos**. 2009. Dissertação (Mestrado em Ciência da Computação) Programa de Pós-graduação em Informática – Núcleo de Computação Eletrônica - Universidade Federal do Rio de Janeiro, Rio de Janeiro, Brasil.

DIANXIANG, X; KENT, M; LIJO, T; LINZHANG, W. **An Automated Test G eneration Technique for *Software* Quality Assurance**. IEEE Transactions on Reliability. v. 64, n. 1 p. 247-268, 2015.

DIAS-NETO, A C**.** Introdução a Teste de *Software*. **Engenharia de *Software* Magazine** n° 1, 2008.

DIJKSTRA, E. **On the Reliability of Mechanisms**. Notes on Structured Programming, EWD249, 2ª Edição, 1970.

EARL, T; HARMAN, M; McMINN, P; SHAHBAZ, M; YOO, S. **The Oracle Problem in *Software* Testing**: A Survey. IEEE Transactions On *Software* Enginnering, v. 41, n. 5, p. 507-525, 2015.

FRANCISCO, M; CASTRO, L. **Automatic Generation of Test Models and Properties from UML Models with OCL Constraints**. Proceeding OCL '12 Proceedings of the 12th Workshop on OCL and Textual Modelling. ACM New York, Innsbruck, Austria, p.49-54, 2012.

GOGOLLA, M; BOHLING, J; RICHTERS, M. **Validating UML and OCL models in USE by automatic snapshot generation**. Journal on Software and System Modeling. Springer. v. 4, n. 4, p.386-398 2005.

GOGOLLA, M; BÜTTNERA, F; RICHTERS, M; **USE: A UML-based specification environment for validating UML and OCL.** Science of Computer Programming. Elsevier North-Holland, Amsterdam. v. 69, n. 1-3, p. 27–34. 2007.

GOGOLLA, M; VALLECILLO, A; BURGUEÑO, L; HILKEN, F. **Employing classifying terms for testing model transformations.** ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems. 2015.

GUIMARÃES, D S; SCHMITZ, E A; LIMA, P; ALENCAR, J A; CORRÊA, A. **A Method for Verifying the Consistency of Business Rules Using Alloy**. SEKE. Pages 381-386. 2014.

ISMAIL, N.; IBRAHIM, R.; IBRAHIM, N. **Automatic Generation of Test Cases from Use-Case Diagram**. Proceedings of the International Conference on Electrical Engineering and Informatics Institut Teknologi: Bandung, Indonesia, 2007.

IEEE 829. **Standard for *Software* Test Documentation** - Description, ANSI/IEEE 829-1998, 1998.

IEEE 829. **Standard for *Software* Test Documentation** - Description, ANSI/IEEE 829-2008, 2008.

JACKSON, D. ***Software*** **Abstractions**: Logic, Language and Analysis. MIT Press, Cambridge, MA, 2006.

JACKSON, D; WING, J. **Lightweight formal methods**. In Hossein Saiedian (ed.), Roundtable Contribution to: An invitation to formal methods, IEEE Computer, v. 29, n. 4, p. 16–30, 1996.

KUHLMANN, M; GOGOLLA M. **From UML and OCL to Relational Logic and Back**. University of Bremen, Database Systems Group, Bremen. 2012.

NAIK, k; TRIPATHY, P. ***Software*** **Testing and Quality Assurance**: Theory and Practice. Wiley. A John Wiley & Sons, inc., Publication, 2008.

NAYAK, A; SAMANTA, D.: **Automatic test data synthesis using UML sequence diagrams**. Journal of Object Technology. v. 9, n. 2 p.115-144. 2010.

OLIVÉ, A. **Conceptual Modeling of Information Systems**. Springer. Berlin. 2007.

OMG, OBJECT MANAGEMENT GROUP. **Object Constraint Language**, fevereiro 2014. Disponível em: <http://www.omg.org/spec/OCL/2.4/PDF>. Acesso em: 10 de outubro de 2016.

PETERS, D; PARNAS, D. **Generating a Test Oracle from Program Documentation**. Proceeding ISSTA '94 Proceedings of the 1994 ACM SIGSOFT international symposium on *Software* testing and analysis . p.58-65. 1994.

PRESSMAN, R. **Engenharia de *Software***: Uma Abordagem Profissional. 7 ed. São Paulo: McGraw-Hill, 2011.

RICHTERS, M. **A precise approach to validating UML models and OCL constraints**. Logos-Verlag University of Bremen, Germany 2002, ISBN 978-3-89722-842-9, pp. 1-210

RICHTERS, M; GOGOLLA, M. **Validating UML models and OCL constraints**, University of Bremen, FB 3, Computer Science Department, Germany. 2005.

SILVA-DE-SOUZA, T. **Uma Abordagem Baseada em Especificação para Teste de Web Services RESTful**. 2012. Dissertação (Mestrado em Informática) - Programa de Pós-graduação em Informática – Núcleo de Computação Eletrônica - Universidade Federal do Rio de Janeiro, Rio de Janeiro, Brasil.

SPILLNER, A; LINZ, T; SCHAEFER, H. ***Software* Testing Foundations**: A Study Guide for the Certified Tester Exam. Foundation Level ISTQB compliant. 4. ed. 2014

SOMMERVILLE, I. **Engenharia de *software***. 9. ed. São Paulo: Pearson, 2011.

SOUSA, H. **Construção Automatizada de Casos de Teste Usando Engenharia Dirigida por Modelos**. 2009. Dissertação (Mestrado em Engenharia de Eletrecidade) – Programa de Pós-Graduação em Engenharia de Eletrecidade – Universidade Federal do Maranhão. Maranhão, Brasil.

USE - A UML based Specification Environment. Preliminary Version 0.1. **Database Systems Group**. Bremen University. 2007

UTTING, M.; LEGEARD, B. **Practical Model-Based Testing:** A Tools Approach. San Francisco: Morgan Kaufmann Publishers Inc., 2007.

WARMER, J; KLEPPE, A. **The Object Constraint Language**, The: Getting Your Models Ready for MDA. 2th. Addison Wesley, 2003.

Apêndice A

A.1 MODELO CONCEITUAL ROYAL&LOYAL

```
model RoyalLoyal

class LoyaltyProgram
attributes
        name: String
operations
end

class Customer
attributes
        name: String
        title: String
        idade: Integer
operations

end

class ProgramPartner
attributes
        numberOfCustomers: Integer
        name: String
operations
end

class ServiceLevel
attributes
        name: String
operations
end

abstract class Transaction
attributes
        points: Integer
operations
end

class Burning < Transaction
end

class Earning < Transaction
end

class CustomerCard
attributes
        valid: Boolean
operations
end

class LoyaltyAccount
attributes
        points: Integer
        number: Integer
operations
end

class Service
attributes
        pointsEarned: Integer
```

```
        pointsBurned: Integer
operations

end

associationclass Membership between
        LoyaltyProgram[*] role programs
        Customer[*] role customers
attributes
operations

end

association LProgramPartner between
        LoyaltyProgram[*] role programs
        ProgramPartner[*] role partners
end

association PPartnerService between
        ProgramPartner[*] role partner
        Service[*] role services
end

association ServiceTransaction between
        Service[*] role services
        Transaction[*] role transactions
end

association LAccountTransaction between
        LoyaltyAccount[*] role account
        Transaction[*] role transactions
end

association CCardTransaction between
        CustomerCard[*] role card
        Transaction[*] role transactions
end

association MembershipLAccount between
        Membership[*] role membership
        LoyaltyAccount[*] role account
end

association MembershipCCard between
        Membership[*] role membership
        CustomerCard[*] role card
end

association CustomerCCard between
        Customer[*] role customer
        CustomerCard[*] role cards
end

association LProgramSLevel between
        LoyaltyProgram[*] role program
        ServiceLevel[*] role levels
end

association MembershipSLevel between
        Membership[*] role membership
        ServiceLevel[*] role currentLevel
end

association ServiceSLevel between
        Service[*] role availableServices
        ServiceLevel[*] role level
End
```

A.2 MODELO CONCEITUAL ABCDEFG

```
model ABCDEFG

class A
attributes
operations
end

class B
attributes
operations
end

class C
attributes
operations
end

class D
attributes
operations
end

class E
attributes
operations
end

class F
attributes
operations
end

class G
attributes
operations
end

association AB between
A[1] role a
B[1] role b
end

association BC between
B[1] role b
C[1] role c
end

association BD between
B[1] role b
D[1] role d
end

association DE between
D[1] role d
E[1] role e
end

association EF between
E[1] role e
F[1] role f
end

association CG between
C[1] role c
G[1] role g
end
```

Apêndice B

B.1 *SCRIPTS* ASSL PARA M1

```
procedure ServiceTests()
var
  service : Service;
begin
        service := Create(Service);
        [service].description := Try([Sequence{'','a', '123456789', '1234567890'}]);
        [service].pointsBurned := Try([Sequence{-1, 0, 1}]);
        [service].pointsEarned := Try([Sequence{-1, 0,1}]);
end;

procedure ServiceLevelTests()
var
  serviceLevel : ServiceLevel;
begin
        serviceLevel := Create(ServiceLevel);
        [serviceLevel].name := Try([Sequence{'','1','123456789','1234567890'}]);
end;

procedure CustomerTests()
var
  customer : Customer;
begin
        customer := Create(Customer);
        [customer].title := Try([Sequence{'','1','123456789','1234567890'}]);
        [customer].name := Try([Sequence{'','1','123456789','1234567890'}]);
end;

procedure CustomerCardTests()
var
  customerCard : CustomerCard;
begin
        customerCard := Create(CustomerCard);
        [customerCard].printedName :=
Try([Sequence{'','a','123456789','1234567890'}]);

end;

procedure LoyaltyAccTests()
var
  loyaltyAccount : LoyaltyAccount;
```

```
begin
        loyaltyAccount := Create(LoyaltyAccount);
        [loyaltyAccount].points := Try([Sequence{0,1}]);
end;

procedure TransactionTests()
var
  burning : Burning,
  earning: Earning;

begin
        burning := Create(Burning);
        earning := Create(Earning);
        [burning].points := Try([Sequence{-1.. 1}]);
        [earning].points := Try([Sequence{-1.. 1}]);
end;

procedure ProgramPartnerTests()
var
  programPartner : ProgramPartner;
begin
        programPartner := Create(ProgramPartner);
        [programPartner].numberOfCustomers := Try([Sequence{-1.. 1}]);
end;

procedure LoyaltyProgramTests()
var
 loyaltyProgram : LoyaltyProgram;
begin
 loyaltyProgram := Create(LoyaltyProgram);
 [loyaltyProgram].name := Try([Sequence{'','a','123456789','1234567890'}]);
end;
```

B2. RESTRIÇÕES OCL SOBRE M1

```
context s:Service
        inv test1: s.description.size > 0 and s.description.size < 10
        inv test2: s.pointsBurned >= 0
        inv test3: s.pointsEarned >= 0

context sl:ServiceLevel
        inv test1: sl.name.size > 0 and sl.name.size < 10

context c:Customer
        inv test1: c.title.size > 0 and c.title.size < 10
        inv test2: c.name.size > 0 and c.title.size < 10
```

```
context cc:CustomerCard
        inv test1: cc.printedName.size > 0 and cc.printedName.size < 10

context la:LoyaltyAccount
        inv test1: la.points >= 0

context t:Transaction
        inv test1: t.points >= 0

context pp:ProgramPartner
        inv test1: pp.numberOfCustomers >= 0
```

B.3 EXEMPLO DE CASO DE TESTE PARA M1

```
-- Input
!new Service
!Service1.description := '123456789'
!Service1.pointsEarned := - 1
!Service1.pointsBurned := 0

-- ExpectedResult -> Invalid state
-- Error inv -> test3
```

Apêndice C

C.1 *SCRIPTS ASSL* PARA M2

```
procedure genSmallScope()
var
  service : Service,
  programPartner : ProgramPartner,
  loyaltyProgram : LoyaltyProgram,
  customer : Sequence(Customer),
  Burning1:  Burning,
  Burning2: Burning,
  loyaltyAccount : LoyaltyAccount;

begin
        service := Create(Service);
        programPartner := Create(ProgramPartner);
        loyaltyProgram := Create(LoyaltyProgram);
        loyaltyAccount := Create(LoyaltyAccount);
        customer := CreateN(Customer, [2]);
        Burning1 := Create(Burning);
        Burning2 := Create(Burning);

        Insert(LProgramPartner, [loyaltyProgram],[programPartner]);
        Insert(PPartnerService,[programPartner],[service]);
        Insert(ServiceTransaction, [service], [Burning1]);
        Insert(ServiceTransaction, [service], [Burning2]);

        for b: Burning in [Burning.allInstances -> asSequence]
        begin
           [b].points := Try([Sequence{4..6}]);
        end;

end;

procedure genSmallScope()
var
  service : Service,
  programPartner : ProgramPartner,
  loyaltyProgram : LoyaltyProgram,
  customer : Sequence(Customer),
  loyaltyAccount : LoyaltyAccount;

begin
```

```
        -- service := Create(Service);
        programPartner := Create(ProgramPartner);
        loyaltyProgram := Create(LoyaltyProgram);
        -- loyaltyAccount := Create(LoyaltyAccount);
        customer := CreateN(Customer, [2]);

        Insert(LProgramPartner, [loyaltyProgram],[programPartner]);

        [programPartner].numberOfCustomers := Try ([Sequence{0..2}]);

        Try(Membership, [LoyaltyProgram.allInstances -> asSequence], [customer]);
end;

procedure genSmallScope()
var
  service : Service,
  programPartner : ProgramPartner,
  loyaltyProgram : LoyaltyProgram,
  customer : Customer,
  loyaltyAccount : LoyaltyAccount;

begin
        service := Create(Service);
        programPartner := Create(ProgramPartner);
        loyaltyProgram := Create(LoyaltyProgram);
        loyaltyAccount := Create(LoyaltyAccount);
        customer := Create(Customer);

        Insert(LProgramPartner,  [loyaltyProgram],[programPartner]);
            Insert(PPartnerService, [programPartner],[service]);
        Insert(Membership, [loyaltyProgram],[customer]);

        [service].pointsBurned := Try([Sequence{0..1}]);
        [service].pointsEarned := Try([Sequence{0..1}]);

        Try(MembershipLAccount, [Membership.allInstances ->
                asSequence],[LoyaltyAccount.allInstances -> asSequence]);

end;

procedure genSmallScope(class1: Integer, class2: Integer)
var
        loyaltyPrograms : Sequence(LoyaltyProgram),
        customer : Customer,
        customerCards: Sequence(CustomerCard);
begin
```

```
        customer:= Create(Customer);

        if[class1 > 0] then
            begin
                loyaltyPrograms := CreateN(LoyaltyProgram, [class1]);
            end;
        if[class2 > 0] then
            begin
                customerCards := CreateN(CustomerCard, [class2]);

                for cc: CustomerCard in [CustomerCard.allInstances ->
                    asSequence]
                    begin
                        [cc].valid := Try([Sequence{true, false}]);
                    end;
            end;
        if[class1 > 0] then
            begin
                Try(Membership, [loyaltyPrograms],[Customer.allInstances ->
                    asSequence]);
            end;
        if[class2 > 0] then
            begin
                Try(CustomerCCard, [Customer.allInstances -> asSequence],
                  [customerCards]);
            end;
    end;
```

C.2 RESTRIÇÕES OCL SOBRE M2

```
    context c: Customer
        inv test2:c.programs -> size = cards -> select(valid = true) -> size

    context lp:LoyaltyProgram
        inv test: lp.partners.services -> forAll(pointsEarned = 0 and pointsBurned = 0)
            implies lp.membership -> select(account -> size > 0)
            -> isEmpty and lp.partners.services
            -> forAll(pointsEarned > 0 or pointsBurned > 0)
            implies lp.membership -> select(account -> size >0) -> notEmpty

    context pp:ProgramPartner
        inv test: pp.numberOfCustomers = pp.programs.customers -> asSet -> size

    context lp:LoyaltyProgram
        inv test: lp.partners.services.transactions -> select(oclIsKindOf(Burning)) ->
collect(points) -> sum < 10.000

    context dt:Customer
        inv test: dt.idade >= 18
```

C.3 EXEMPLO DE CASO DE TESTE GERADO PARA M2

```
-- input
!new Customer
!new CustomerCard
!new LoyaltyProgram
!insert (Customer1,CustomerCard1) into CustomerCCard
!CustomerCard1.valid := false

-- ExpectedResult -> Invalid state
-- Error inv -> test2
```

Apêndice D

D.1 *SCRIPTS* ASSL PARA M3

```
procedure preEnroll1()
var customer : Customer;
begin
        customer := Create(Customer);
        [customer].id := [1];
end;

procedure preEnroll2()
var customer : Customer;
begin
        customer := Create(Customer);
        [customer].id := [2];
end;

procedure Enroll(customer : Customer)
var programPartner: ProgramPartner,
        loyaltyProgram: LoyaltyProgram,
        serviceLevel: ServiceLevel,
        loyaltyAccount: LoyaltyAccount,
        card : CustomerCard,
        service: Service;

begin
        -- customer := Create(Customer);
        customer.[name] := Try([Sequence{'','a','abcdfdswef', 'adehjklçpo'}]);
        customer.[title] := Try([Sequence{'','a','abcdfdswef', 'adehjklçpo'}]);
        customer.[idade] := Try([Sequence{17,18,19}]);

        programPartner := Create(ProgramPartner);
        loyaltyProgram := Create(LoyaltyProgram);
        serviceLevel := Create(ServiceLevel);
        service := Create(Service);
        loyaltyAccount := Create(LoyaltyAccount);

        Try(Membership, [LoyaltyAccount.allInstances ->
                                asSequence],[customer]);

        Insert(CustomerCCard, [customer],[card]);
        Insert(MembershipLAccount, [membership], [loyaltyAccount]);
        Insert(MembershipSLevel, [membership], [serviceLevel]);
```

```
        Insert(MembershipCCard, [membership], [card]);

end;

procedure preUnroll()
var
        loyaltyProgram : LoyaltyProgram,
        customer : Customer;
begin
        loyaltyProgram := Create(LoyaltyProgram);
        customer := Create(Customer);
        [customer].id := 1;

        Insert(Membership, [loyaltyProgram],[customer]);

end;

procedure Unroll(membership: Customer)
var post : Boolean;
begin
        if[membership.customer.id = 1] then
                begin
                        Delete(membership);
                end;
        else
                begin
                        post := False
                end;
end;

procedure preGenNewCards()
var
        loyaltyProgram : LoyaltyProgram,
        customer : Customer;

begin
        loyaltyProgram := Create(LoyaltyProgram);
        customer := Create(Customer);
        [customer].id := 1;

        Insert(Membership, [loyaltyProgram],[customer]);

end;

procedure genNewCards(customer: Customer)
var
        customerCard : CustomerCard;
```

```
        cards : Integer,
        customerCard: CustomerCard;
begin
 cards := Try([Sequence{0..1}]);
 if[cards = 1] then
        begin
                customerCard := Create(customerCard);
        end;
 else
        begin
                customerCard := Create(CustomerCard);
                [customerCard].id := 1;

                Insert(CustomerCCard, [customer],[customerCard]);
                Insert(MembershipCCard, [customer.membership],[customerCard]);
        end;
end;
```

D.2 RESTRIÇÕES OCL SOBRE M3

```
context lp:LoyaltyProgram
 inv pre_enroll: lp.customers -> select(id = 1) -> size = 0
 inv post_enroll: lp.customers -> includes(Customer1)

context lp:LoyaltyProgram
inv pre_unroll: lp.customers -> select(id=1) -> size > 0
inv post_unroll: lp.customers -> select(id=1) -> size = 1

context cc:Customer
inv pre_generateCards: cc.cards -> select(id=1) -> size = 0
inv post_generateCards: cc.cards -> select(id=1) -> size = 1
inv test1: cc.idade >= 18
inv test2: c.title.size > 0 and c.title.size < 10
inv test3: c.name.size > 0 and c.title.size < 10
```

D.3 EXEMPLO DE CASO DE TESTE PARA M3

```
-- Input
!new Customer
!Customer1.id := 1
!Customer1.idade := 17
!Customer1.name := 'adehjklçpo'
!Customer1.title := 'a'
!new ProgramPartner
!new LoyaltyProgram
!new ServiceLevel
```

```
!new LoyaltyAccount
!new CustomerCard
!new Service

!insert (LoyaltyProgram1,Customer1) into Membership
!insert (Service1,ServiceLevel1) into ServiceSLevel
!insert (ProgramPartner1,Service1) into PPartnerService
!insert (Customer1,CustomerCard1) into CustomerCCard
!insert (Membership1,LoyaltyAccount1) into MembershipLAccount
!insert (Membership1,ServiceLevel1) into MembershipSLevel

-- ExpectedResult -> Invalid State
-- Error inv -> test1, test3
```

Apêndice E

E.1 *SCRIPTS* ASSL PARA M4

```
procedure genSmallScope(class1: Integer, class2: Integer)
var
  burnings: Sequence(Burning),
  earnings: Sequence(Earning);

begin
  if[class1 > 0] then
        begin
          burnings := CreateN(Burning, [class1]);
        end;

  if[class2 > 0] then
        begin
          earnings := CreateN(Earning, [class2]);
          --Try(ServiceTransaction, [Service.allInstances -> asSequence],[earnings]);
          --Try(LAccountTransaction, [LoyaltyAccount.allInstances ->
asSequence],[earnings]);
          --Try(CCardTransaction, [CustomerCard.allInstances ->
asSequence],[earnings]);
        end;

  if[class1 > 0] then
        begin
          Try(ServiceTransaction, [Service.allInstances -> asSequence],[burnings]);
          Try(LAccountTransaction, [LoyaltyAccount.allInstances ->
asSequence],[burnings]);
          Try(CCardTransaction, [CustomerCard.allInstances ->
asSequence],[burnings]);
        end;

  if[class2 > 0] then
        begin
          Try(ServiceTransaction, [Service.allInstances -> asSequence],[earnings]);
          Try(LAccountTransaction, [LoyaltyAccount.allInstances ->
asSequence],[earnings]);
          Try(CCardTransaction, [CustomerCard.allInstances ->
asSequence],[earnings]);
        end;
end;
```

```
        procedure genSmallScope(class1: Integer, class2: Integer, class3: Integer, class4:
Integer)
        var
          services: Sequence(Service),
          programPartners: Sequence(ProgramPartner),
          loyaltyPrograms: Sequence(LoyaltyProgram),
          serviceLevels: Sequence(ServiceLevel);

        begin
          if [class1 > 0] then
                begin
                  services := CreateN(Service, [class1]);
                end;

          if [class2 > 0] then
                begin
                  programPartners:= CreateN(ProgramPartner, [class2]);
                end;

          if [class3 > 0] then
                begin
                  loyaltyPrograms:= CreateN(LoyaltyProgram, [class3]);
                end;

          if [class4 > 0] then
                begin
                  serviceLevels:= CreateN(ServiceLevel, [class4]);
                end;

          if[class1 > 0 and class2 > 0] then
                begin
                  Try(PPartnerService, [programPartners],[services]);
                end;

          if[class1 > 0 and class4 > 0] then
                begin
                  Try(ServiceSLevel, [services], [serviceLevels]);
                end;

          if[class2 > 0 and class4 > 0] then
                begin
                  Try(LProgramPartner, [loyaltyPrograms],[programPartners]);
                end;

          if[class4 > 0 and class3 > 0] then
                begin
                  Try(LProgramSLevel, [loyaltyPrograms],[serviceLevels]);
```

```
            end;
        end;

        procedure preSetup()
        var
          loyaltyProgram1: LoyaltyProgram,
          loyaltyProgram2: LoyaltyProgram;

        begin
          loyaltyProgram1 := Create(LoyaltyProgram);
          loyaltyProgram2 := Create(LoyaltyProgram);
        end;

        procedure genSmallScope(class1: Integer, class2: Integer)
        var
          customerCards : Sequence(CustomerCard),
          customers : Sequence(Customer);

        begin
          if[class1 > 0] then
            begin
              customers := CreateN(Customer, [class1]);
            end;

          if[class2 > 0] then
            begin
              customerCards := CreateN(CustomerCard, [class2]);
            end;

          if[class1 > 0 and class2 > 0] then
            begin
              Try(CustomerCCard, [customers],[customerCards]);
            end;

          if[class1 > 0 and LoyaltyProgram.allInstances -> size > 0] then
            begin
              Try(Membership,[LoyaltyProgram.allInstances -> asSequence], [customers]);
            end;

          if[Membership.allInstances -> size > 0 and class2 > 0] then
            begin
              Try(MembershipCCard, [Membership.allInstances -> asSequence],
[customerCards]);
            end;
        end;

        procedure preSetup()
```

```
var
  loyaltyProgram1: LoyaltyProgram,
  loyaltyProgram2: LoyaltyProgram,
  serviceLevel1 : ServiceLevel,
  serviceLevel2 : ServiceLevel,
  loyaltyAccount1 : LoyaltyAccount,
  loyaltyAccount2 : LoyaltyAccount,
  customerCard2 : CustomerCard,
  customerCard1 : CustomerCard,
  customer1 : Customer,
  customer2 : Customer;


begin
  loyaltyProgram1 := Create(LoyaltyProgram);
  loyaltyProgram2 := Create(LoyaltyProgram);

  serviceLevel1 := Create(ServiceLevel);
  serviceLevel2 := Create(ServiceLevel);

--loyaltyAccount1 := Create(LoyaltyAccount);
-- loyaltyAccount2 := Create(LoyaltyAccount);

  customerCard1 := Create(CustomerCard);
  customerCard2 := Create(CustomerCard);

  customer1 := Create(Customer);
  customer2 := Create(Customer);

  Insert(Membership, [loyaltyProgram1],[customer1]);
  Insert(Membership, [loyaltyProgram2],[customer1]);
  -- Insert(Membership, [loyaltyProgram2],[customer2]);
  Insert(CustomerCCard, [customer1],[customerCard1]);
  Insert(CustomerCCard, [customer2],[customerCard2]);
end;

procedure genSmallScope(class1: Integer)
var
  loyaltyAccounts : Sequence(LoyaltyAccount);

begin

  if[class1 > 0] then
          begin
            loyaltyAccounts := CreateN(LoyaltyAccount, [class1]);
           end;
```

```
        if[class1 > 0] then
                begin
                  Try(MembershipLAccount, [Membership.allInstances ->
asSequence],[loyaltyAccounts]);
                end;

        Try(MembershipCCard, [Membership.allInstances -> asSequence],
[CustomerCard.allInstances -> asSequence]);
        Try(MembershipSLevel, [Membership.allInstances -> asSequence],
[ServiceLevel.allInstances -> asSequence]);

        end;
```

E.2 REGRAS DE MULTIPLICIDADE COMO RESTRIÇÕES OCL (M4)

```
        context b: Burning
                inv ServiceRuleBurning: b.services -> size =  1
                inv LoyaltyAccountRuleBurning: b.account -> size() = 1
                inv CustomerCardRuleBurning: b.card -> size() = 1

        context e: Earning
                inv ServiceRuleEarning: e.services -> size =  1
                inv LoyaltyAccountRuleEarning: e.account -> size() = 1
                inv CustomerCardRuleEarning: e.card -> size() = 1

        context s:Service
                inv ManyServicesOnlyOneProgramPartner:
                  s.partner -> size = 1

        context pp: ProgramPartner
                inv OneProgramPartnerOneManyLoyaltyAcc:
                  pp.programs -> size > 0

        context lp: LoyaltyProgram
                inv OneLoyaltyProgramOneManyProgramPartner:
                  lp.partners -> size > 0

                inv OneLoyaltyAccOneManyServiceLeves:
                  lp.levels -> size > 0

        context sl: ServiceLevel
                inv ManyServiceLevelsOneLoyaltyProg:
                  sl.program -> size = 1

                context cc: CustomerCard
```

```
        inv OnlyOneCustomer: cc.customer -> size = 1
        inv OnlyOneMember: cc.membership -> size = 1

context m: Membership
        inv OnlyOneCard: m.card -> size = 1
        inv OnlyServiceLevel: m.currentLevel -> size = 1
        inv MaxOneAccount: m.account -> size < 2

context la: LoyaltyAccount
        inv OnlyOneMembership: la.membership -> size = 1
```

E.3 EXEMPLO DE CASO DE TESTE PARA M4

```
-- Input
!new Burning
!new Service
!new LoyaltyAccount
!new CustomerCard
!insert (Service1,Burning1) into ServiceTransaction
!insert (LoyaltyAccount1,Burning1) into LAccountTransaction

-- ExpectedResult -> Invalid state
-- Error inv -> CustomerCardRuleBurning
```

Apêndice F

F.1 *SCRIPTS* ASSL DO QUASE-EXPERIMENTO

```
procedure genSmallScope1(aInt: Integer, bInt: Integer)
var aObjs: Sequence(A), bObjs: Sequence(B);
begin
 if[aInt > 0] then
   begin
     aObjs := CreateN(A,[aInt]);
   end;

 if[bInt > 0] then
   begin
     bObjs := CreateN(B,[bInt]);
   end;

 if[aInt > 0 and bInt > 0] then
   begin
     Try(AB, [aObjs],[bObjs]);
   end;
end;

procedure genSmallScope2(aInt: Integer, bInt: Integer, cInt: Integer)
var aObjs: Sequence(A), bObjs: Sequence(B), cObjs: Sequence(C);
begin
 if[aInt > 0] then
   begin
     aObjs := CreateN(A,[aInt]);
   end;

 if[bInt > 0] then
   begin
     bObjs := CreateN(B,[bInt]);
   end;

 if[cInt > 0] then
   begin
     cObjs := CreateN(C,[cInt]);
   end;

 if[aInt > 0 and bInt > 0] then
   begin
     Try(AB, [aObjs],[bObjs]);
```

```
    end;

  if[bInt > 0 and cInt > 0] then
    begin
      Try(BC, [bObjs],[cObjs]);
    end;
  end;

  procedure genSmallScope3(aInt: Integer, bInt: Integer, cInt: Integer, dInt: Integer)
  var aObjs: Sequence(A), bObjs: Sequence(B), cObjs: Sequence(C), dObjs:
Sequence(D);
  begin
   if[aInt > 0] then
    begin
      aObjs := CreateN(A,[aInt]);
    end;

   if[bInt > 0] then
    begin
      bObjs := CreateN(B,[bInt]);
    end;

   if[cInt > 0] then
    begin
      cObjs := CreateN(C,[cInt]);
    end;

   if[dInt > 0] then
    begin
      dObjs := CreateN(D,[dInt]);
    end;

   if[aInt > 0 and bInt > 0] then
    begin
      Try(AB, [aObjs],[bObjs]);
    end;

   if[bInt > 0 and cInt > 0] then
    begin
      Try(BC, [bObjs],[cObjs]);
    end;

   if[bInt > 0 and dInt > 0] then
    begin
      Try(BD, [bObjs],[dObjs]);
    end;
  end;
```

```
        procedure genSmallScope4(aInt: Integer, bInt: Integer, cInt: Integer, dInt: Integer,
eInt: Integer)
        var aObjs: Sequence(A), bObjs: Sequence(B), cObjs: Sequence(C), dObjs:
Sequence(D), eObjs: Sequence(E);
        begin
         if[aInt > 0] then
          begin
           aObjs := CreateN(A,[aInt]);
          end;

         if[bInt > 0] then
          begin
           bObjs := CreateN(B,[bInt]);
          end;

         if[cInt > 0] then
          begin
           cObjs := CreateN(C,[cInt]);
          end;

         if[dInt > 0] then
          begin
           dObjs := CreateN(D,[dInt]);
          end;

         if[eInt > 0] then
          begin
           eObjs := CreateN(E,[eInt]);
          end;

         if[aInt > 0 and bInt > 0] then
          begin
           Try(AB, [aObjs],[bObjs]);
          end;

         if[bInt > 0 and cInt > 0] then
          begin
           Try(BC, [bObjs],[cObjs]);
          end;

         if[bInt > 0 and dInt > 0] then
          begin
           Try(BD, [bObjs],[dObjs]);
          end;

         if[dInt > 0 and eInt > 0] then
```

```
    begin
      Try(DE, [dObjs],[eObjs]);
    end;
  end;

  procedure genSmallScope5(aInt: Integer, bInt: Integer, cInt: Integer, dInt: Integer,
eInt: Integer, fInt: Integer)
  var aObjs: Sequence(A), bObjs: Sequence(B), cObjs: Sequence(C), dObjs:
Sequence(D), eObjs: Sequence(E), fObjs: Sequence(F);
  begin
   if[aInt > 0] then
    begin
      aObjs := CreateN(A,[aInt]);
    end;

   if[bInt > 0] then
    begin
      bObjs := CreateN(B,[bInt]);
    end;

   if[cInt > 0] then
    begin
      cObjs := CreateN(C,[cInt]);
    end;

   if[dInt > 0] then
    begin
      dObjs := CreateN(D,[dInt]);
    end;

   if[eInt > 0] then
    begin
      eObjs := CreateN(E,[eInt]);
    end;

   if[fInt > 0] then
    begin
      fObjs := CreateN(F,[fInt]);
    end;

   if[aInt > 0 and bInt > 0] then
    begin
      Try(AB, [aObjs],[bObjs]);
    end;

   if[bInt > 0 and cInt > 0] then
    begin
```

```
    Try(BC, [bObjs],[cObjs]);
   end;

  if[bInt > 0 and dInt > 0] then
   begin
    Try(BD, [bObjs],[dObjs]);
   end;

  if[dInt > 0 and eInt > 0] then
   begin
    Try(DE, [dObjs],[eObjs]);
   end;

  if[eInt > 0 and fInt > 0] then
   begin
    Try(EF, [eObjs],[fObjs]);
   end;
 end;

 procedure genSmallScope6(aInt: Integer, bInt: Integer, cInt: Integer, dInt: Integer,
eInt: Integer, fInt: Integer, gInt: Integer)
 var aObjs: Sequence(A), bObjs: Sequence(B), cObjs: Sequence(C), dObjs:
Sequence(D), eObjs: Sequence(E), fObjs: Sequence(F), gObjs: Sequence(G);
 begin
  if[aInt > 0] then
   begin
    aObjs := CreateN(A,[aInt]);
   end;

  if[bInt > 0] then
   begin
    bObjs := CreateN(B,[bInt]);
   end;

  if[cInt > 0] then
   begin
    cObjs := CreateN(C,[cInt]);
   end;

  if[dInt > 0] then
   begin
    dObjs := CreateN(D,[dInt]);
   end;

  if[eInt > 0] then
   begin
    eObjs := CreateN(E,[eInt]);
```

```
  end;

 if[fInt > 0] then
  begin
   fObjs := CreateN(F,[fInt]);
  end;

 if[gInt > 0] then
  begin
   gObjs := CreateN(G,[gInt]);
  end;

 if[aInt > 0 and bInt > 0] then
  begin
   Try(AB, [aObjs],[bObjs]);
  end;

 if[bInt > 0 and cInt > 0] then
  begin
   Try(BC, [bObjs],[cObjs]);
  end;

 if[bInt > 0 and dInt > 0] then
  begin
   Try(BD, [bObjs],[dObjs]);
  end;

 if[dInt > 0 and eInt > 0] then
  begin
   Try(DE, [dObjs],[eObjs]);
  end;

 if[eInt > 0 and fInt > 0] then
  begin
   Try(EF, [eObjs],[fObjs]);
  end;

 if[cInt > 0 and gInt > 0] then
  begin
   Try(CG, [cObjs],[gObjs]);
  end;
end;
```

F.2 REGRAS DE MULTIPLICIDADE COMO RESTRIÇÕES OCL

```
-- 1 relacionamento
context a:A
```

```
 inv relA: a.b -> size = 1

context b:B
 inv relB: b.a -> size = 1

-- 2 relacionamentos
context a:A
 inv relAB: a.b -> size = 1

context b:B
 inv relBA: b.a -> size = 1
 inv relBC: b.c -> size = 1

context c:C
 inv relCB: c.b -> size = 1

-- 3 relacionamentos
context a:A
 inv relAB: a.b -> size = 1

context b:B
 inv relBA: b.a -> size = 1
 inv relBC: b.c -> size = 1
 inv relBD: b.d -> size = 1

context c:C
 inv relCB: c.b -> size = 1

context d:D
 inv relDB: d.b -> size = 1

-- 4 relacionamentos
context a:A
 inv relAB: a.b -> size = 1

context b:B
 inv relBA: b.a -> size = 1
 inv relBC: b.c -> size = 1
 inv relBD: b.d -> size = 1

context c:C
 inv relCB: c.b -> size = 1

context d:D
 inv relDB: d.b -> size = 1
 inv relDE: d.e -> size = 1
```

```
context e:E
 inv relED: e.d -> size = 1

-- 5 relacionamentos
context a:A
 inv relAB: a.b -> size = 1

context b:B
 inv relBA: b.a -> size = 1
 inv relBC: b.c -> size = 1
 inv relBD: b.d -> size = 1

context c:C
 inv relCB: c.b -> size = 1

context d:D
 inv relDB: d.b -> size = 1
 inv relDE: d.e -> size = 1

context e:E
 inv relED: e.d -> size = 1
 inv relEF: e.f -> size = 1

context f:F
 inv relFE: f.e -> size = 1

-- 6 relacionamentos
context a:A
 inv relAB: a.b -> size = 1

context b:B
 inv relBA: b.a -> size = 1
 inv relBC: b.c -> size = 1
 inv relBD: b.d -> size = 1

context c:C
 inv relCB: c.b -> size = 1
 inv relCG: c.g -> size = 1

context d:D
 inv relDB: d.b -> size = 1
 inv relDE: d.e -> size = 1

context e:E
 inv relED: e.d -> size = 1
 inv relEF: e.f -> size = 1
```

```
context f:F
 inv relFE: f.e -> size = 1

context g:G
 inv relGC: g.c -> size = 1
```